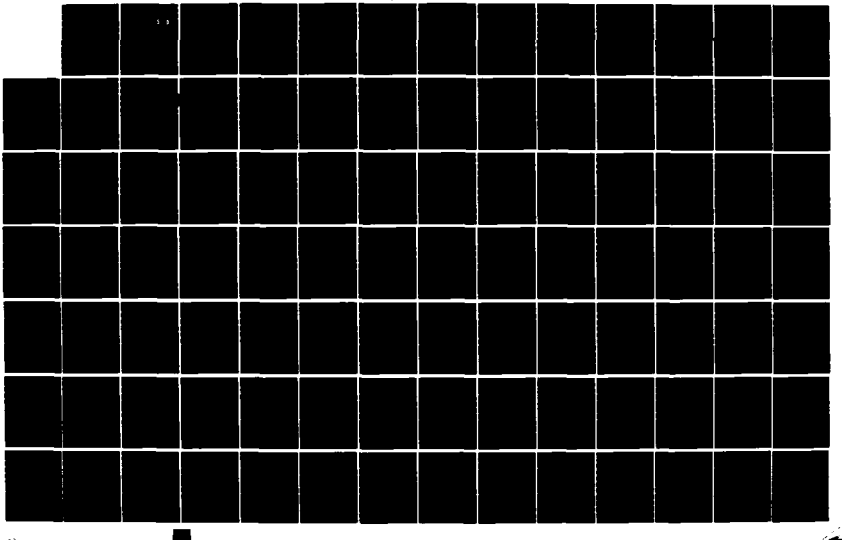
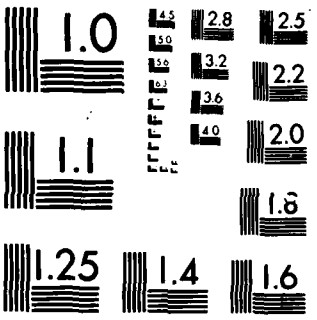
AD-A164 022 KNOWLEDGE-BASED TRANSFORMATIONAL SYNTHESIS OF EFFICIENT 1/
STRUCTURES FOR CO.. (U) KESTREL INST PALO ALTO CA
R M KING 30 SEP 85 KES. U. 85. 5 AFOSR-TR-85-1259
UNCLASSIFIED F49620-85-C-0015 F/G 9/2 NL





(2)

KES.U.85.5

Kestrel Institute

Knowledge-Based Transformational Synthesis of Efficient Structures for Concurrent Computation

by RICHARD M. KING

May 1985

Research sponsored by the Air Force Office of Scientific Research (AFOSR), United States Air Force, under contract F49620-85-C-0015. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

1801 Page Mill Road, Palo Alto, California 94304 (415) 493-6871

Uncl

SECURITY CLASSIFICATION OF THIS PAGE

AD-A164022

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S): KES.U.85.5	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 85-1259
6a. NAME OF PERFORMING ORGANIZATION: Kestrel Institute; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: same as #8
6c. ADDRESS (City, State and ZIP Code): 1801 Page Mill Road, Palo Alto, CA. 94304	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-85-C-0015
8c. ADDRESS (City, State and ZIP Code): Bolling AFB Washington, D.C. 20332	10. SOURCE OF FUNDING NOS: PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO A2; WORK UNIT NO.

11. TITLE (Include Security Classification)
Knowledge-Based Transformational Synthesis of Efficient Structures for Concurrent Computation

12. PERSONAL AUTHOR(S)
Richard M. King

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final Technical	FROM 10/84 TO 9/85	1985 September 30	195

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Multi-processor synthesis; Tree-structured multiprocessors; Concurrency; Closures; Divide and conquer; Trees; Actors

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The object of our research is the codification of programming knowledge for the synthesis of concurrent programs. This is important because concurrency is a way of securing better performance on amenable problems than is available on non-concurrent computers. We divide this knowledge into two sections: knowledge for the synthesis of arrays of processors that could be connected in a geometrically regular manner (crystalline concurrency), and knowledge for the synthesis of tree structures (tree concurrency). We divide synthesis of crystalline concurrency, in turn, into several subsections: synthesis of declarations of multiple processors and the wires implied by the dependencies among the values they contain, reduction of this wire network to a smaller wire network, creation of subnetworks to replace an overly-broad fanout network, *virtualization* which is the creation of additional array elements and processors to reflect the internal enumerations that comprise the computation of a datum, and *aggregation* which is the merging of several processors into one. We use a transformational approach. The transformational system has *rules*, each of which contains two predicates: an *antecedent* and a *consequent*. If the antecedent of a rule is true of a given object, the rule *applies* and the object is modified to make the consequent true.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED UNLIMITED X SAME AS RPT ☐ DTIC USERS ☐	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
[redacted]	(202) 767-[redacted]	[redacted] NM

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE

Uncl
SECURITY CLASSIFICATION OF THIS PAGE

ABSTRACT

Knowledge-Based Transformational Synthesis
of Efficient Structures for Concurrent Computation

by RICHARD M. KING

The object of our research is the codification of programming knowledge for the synthesis of concurrent programs. This is important because concurrency is a way of securing better performance on amenable problems than is available on non-concurrent computers. We divide this knowledge into two sections: knowledge for the synthesis of arrays of processors that could be connected in a geometrically regular manner (crystalline concurrency), and knowledge for the synthesis of tree structures (tree concurrency).

We divide synthesis of crystalline concurrency, in turn, into several subsections: synthesis of declarations of multiple processors and the wires implied by the dependencies among the values they contain, reduction of this wire network to a smaller wire network, creation of subnetworks to replace an overly-broad fanout network, *virtualization* which is the creation of additional array elements and processors to reflect the internal enumerations that comprise the computation of a datum, and *aggregation* which is the merging of several processors into one.

We also divide tree concurrency synthesis. Our primary technique is divide and conquer, but to make this technique effective we must take another view of the specification. We respecify a given requirement, that of computing a new array whose values are pointwise computable as a function of an existing array and an index, as a requirement to compute a functional object whose side effect is to satisfy the original specification, together with the requirement that said object be called with the proper arguments. We call the computed functional object a *closure*.

We use a transformational approach. The transformational system has *rules*, each of which contains two predicates: an *antecedent* and a *consequent*. If the antecedent of a rule is true of a given object, the rule *applies* and the object is modified to make the consequent true.

We demonstrate these techniques' ability to synthesize one or more solutions to each of several classical problems from the literature. These solutions are topological descriptions of arrays of computing elements ("processors"). The resulting elements' complexities range from a couple of gates to something comparable to a microprocessor. We do not attempt to actually lay out the processors and interconnections.

Acknowledgements

I would like to thank all of the following:

- My wife, Rebecca, for being herself during this project,
- Robert Paige, for advising me on various matters and for introducing me to Ernst Mayr,
- The Kestrel Institute, for providing me with an economically and intellectually supportive environment for completing a Ph. D. thesis,
- Professor Ernst Mayr; for providing guidance on this work, for reading innumerable rough drafts and for holding me to high standards,
- Cordell Green, for maintaining the supportiveness of the Kestrel Institute and for criticizing some of my ideas and writing style,
- Lydia Skinner and Maria Pryce, for reading the "final" draft one last time and finding a surprising number of typographical errors,
- Gordy Kotik, Tom Pressburger, Doug Smith, Steve Westfold, and the rest of the gang at the Kestrel Institute, for listening patiently but critically whenever I got an inspiration and helping me to squeeze out the junk, and
- Janet Willis and Elsie Jackson, for helping me with the logistics of a 4500 kilometer separation.

Were it not for these forms of support you would be reading somebody else's thesis.

Contents

List of Illustrations

Chapter 1

Introduction and Approach

Computation power can be delivered in several denominations, ranging from chips that can execute a few hundred eight-bit instructions per second up to large-scale computers that can approach a hundred million 64-bit instructions per second. There is no reason to suppose that the larger computers deliver more computation power per dollar than the smaller ones. At present, cheap computer power seems to come in small packages. We can informally argue that a given problem requires a certain number of gates for a solution in a given amount of time, whether these gates are contained in a single large package or several small ones. Additionally, in the large processor there are constraints that force use of extra logic to keep track of some of the work that the machine is trying to overlap with other related work. Some of those gates may, at times, lie fallow, as may some computational gates in the large machine if the mix of tasks is instantaneously different from that which the designer assumed during construction. Additionally, we argue that the electronics industry produces a much greater volume of computation in small chunks than of computation in large chunks, and therefore enjoys the economics of scale.

We could accept slow but cheap computation, allowing us to afford many computers. There are situations where this is the proper course of action. An arcade owner does better to have a small computer in each game of his arcade instead of a single large processor to power all of the games, even ignoring the reliability and engineering problems of the latter

approach. Companies are beginning to supply each of their employees with a desk-top computer rather than with a terminal into a large computing system.

Slow computation is not, however, acceptable in all cases. There are classes of situations in which a certain minimum amount of computer power must be provided. These situations range from fast real-time systems such as avionics, through situations such as weather forecasting where we might attempt to model 48 hours of atmospheric behavior in 24 hours, where more computation power would allow a finer-grained model of the atmosphere, to aerodynamic modeling where an increase in computer power improves the situation from the point where it's better to experiment on actual hardware to the point where it is better to experiment on computer models.

1.1 The Need for a VLSI Design Assistant

The number of devices that can fit on an integrated circuit continues to increase. It is expected [MeC80] that there will be at least one more factor of ten reduction in the feature size of integrated circuits before physical limits are reached, giving a hundredfold increase in the number of devices that can be integrated on a chip of a given size. Additionally, it would not be unreasonable to expect some increase in the maximum size of chips.

At present it is practical for a designer to specify all of the functional blocks of his design. Current technology allows for a number of gates on a chip approaching a million, but computer aided design tools can allow him to deal with the complexity that this allows by specifying circuit information as logic diagrams rather than as circuit masks, and in some new systems a more convénient form than logic diagrams is used (for example in MACPITTS [SSC82] the design is specified in a LISP-like language). Still, the entire functional design comes from the designer.

In the future it will be possible to squeeze a hundred times as much function on a chip as is now possible. Good ways must be found to exploit this capability and to create

chips which make good use of a hundred times as many gates as current chips have. One technique would be to have a number of functional blocks on a chip comparable to the number in current designs. Most blocks will have to be larger than those of current chips in order to put as much functionality on a chip as will be possible without tremendously increasing the number of blocks.

One method for allowing these larger blocks is to have a library of large blocks available to the designer. This would be undesirable because it is plausible that the number of blocks desired by various designers is a rapidly growing function of their size. For example, since the sizes of functional blocks would be at least comparable to the size of current chips, and since one of the major constraints on modern chip designs (pin limitations) wouldn't apply, it would seem reasonable to suppose that there should be at least as many functional blocks available as there are chip types now. This would be unacceptable.

Another approach is to use hierarchical design methods. In effect each designer creates an *ad hoc* library. This approach has the problems inherent in private subroutine libraries, including difficulties of sharing effort in a large project.

We have a fairly close analogy between the future situation regarding the ability of VLSI fabricators to make large chips and the current ability of software designers to "fabricate" software. Language designers can either provide more primitive operations and therefore hope to cover the needs of programmers, or they can provide the programmer with the option of building his own large building blocks out of smaller ones. These are called subroutines. Designers of languages like COBOL, APL and SNOBOL attempted to provide numerous primitive operations and precisely the correct ones for certain problem domains (although they felt it necessary to provide the ability to define subroutines as well). Other languages such as LISP provide few primitive operations but are intended to facilitate creation of subroutines. Limits to this approach have been recognized. See, for example, [**Sch80**] in which it is pointed out that modularity facilitates software construction at

the expense of efficiency, and [DoD83] in which it is pointed out that currently proposed software tasks not only cannot be done efficiently but cannot even be done reliably without using something more advanced than modular techniques.

At our laboratory [GCP81] and others [Hor81] work is proceeding on knowledge based software assistants, which are systems that allow their user to describe a desired system behavior is a specification language. This thesis describes the beginning of a knowledge based VLSI designer assistant.

1.2 Goals

Several steps must be take to exploit the cost-effectiveness of small denominations of computation:

- **The processors need to communicate.** If a system has a large number of small processors and no wires between them the only thing it can do is solve a large number of small problems simultaneously. This is not always an accurate reflection of what people want. The nature of the problems that the processors will be able to solve quickly depends critically on the way they are interconnected. Unfortunately, so does the cost of the multiprocessor system, and it turns out that the most versatile topologies are those that cost the most to wire.

- **The processors need to be scheduled.** Somebody must decide what each processor will do when. One can not merely take an algorithm that is carefully crafted to run on a single large processor and make it run on a multiprocessor system with little change. In [AMa84] it is shown that some problems can be solved by efficient single-processor algorithms that have *no* concurrent analog. There are concurrent solutions to the original problem, but the sequential algorithm produces a *specific* solution. No algorithm, sequential or concurrent, can solve the problem of finding

the specific solution that the sequential algorithm would have produced faster than in polynomial time (unless $\mathcal{P} \subseteq \mathcal{NC}$ where (for problem instances of size n) $\mathcal{P}$ is the set of problems solvable in $O(n^i)$ time (for some constant i) on a single processor and $\mathcal{NC}$ is the set of problems solvable in $O(\log^i)$ time using $O(n^j)$ processors (for constants i and j). It is believed that $\mathcal{P} \not\subseteq \mathcal{NC}$.).

- **The processors need to be loaded.** Normally some entity outside of the problem-solving computer presents the data representing a problem instance at a single source, and the computer is required to deliver the results to a single destination. The program would also normally be stored on a single device for economic reasons. We must address the issues of how instruction loading, input and output will take place across the single-stream/multiple-stream interface.

We use the approach of synthesizing a concurrent version of a specification expressed in an extremely "high style". The reason for this is that specifications are turned into programs by the addition of specialized information (such as data structure selection), and the removal of this specialized information is difficult. This information imposes constraints on the manner in which the calculation is carried out, and these constraints make it more difficult to produce a program optimal according to one set of criteria from another program that was (previously) optimized for another set.

Our system, which we call TRANSCONS (for the TRANSformational CONcurrency Synthesizer), accepts input/output specifications in a high level language. It transforms these into descriptions of parallel structures.

We will not concern ourselves with the placement of processing nodes or gates on a surface or in space, even though the quality of the placement can drastically affect efficiency by altering wire lengths and therefore path delays and costs. We feel that our system meets its need despite this omission for the following reasons:

- the topology could be adequate for our needs
- systems for laying out circuits, given the topologies, exist
- the topologies that TRANSCONS synthesizes have fairly obvious layouts (i.e., the coordinates of the position of a processor are linear functions of its indices)

One possible use of the output of a TRANSCONS run is to control the operation of a "universal" parallel computer such as a shuffle exchange or cube-connected cycle system. Such a use of TRANSCONS output might be made for testing purposes, but the expense of the universal parallel computers and the $O(\log n)$ factor speed loss for simulation of an n-processor system with direct interconnections as specified by TRANSCONS make it unlikely that this will be a standard use of this technology.

Universal architectures described above all have the unattractive property that some of the wires must be long, and the total wire length is long. They also have some extremely untidy wiring layouts in any physical implementation (necessarily so; since every surface that bisects the network must be pierced by a large number of wires, the wiring arrangement contains few bundles of wires tracing adjacent paths.). Wiring is one of the least reliable parts of a modern digital computer system. In addition, the $\log n$ speed factor can be a serious matter.

It would therefore be desirable in some cases to reduce the average length of the wires and increase the orderliness of the interconnections, so intermodule connections can be reduced to interboard interconnections, which can be reduced to printed circuit connections and in turn to connections within a chip. This reduces cost and increases reliability.

The techniques of TRANSCONS produce topologies that could easily be laid out by computer as tidy layouts. This is because the various expressions controlling the interconnections between the processors are restricted to be from Presburger Arithmetic, and they are simple expressions linear in the processors' names.

If TRANSCONS produces a crystalline interconnection pattern with higher dimensionality than an available network, it is still possible to find an assignment of logical processors to physical ones that incurs only a moderate speed penalty. For example, there is a simple mapping of a $\sqrt[3]{n} \times \sqrt[3]{n} \times \sqrt[3]{n}$ array of processors onto a two dimensional array such that the cost of communication along two of the simulated dimensions is $O(1)$ and that of the third is $O(\sqrt[3]{n})$. The constant factor would perhaps be one third of that of a simulation on a universal computer, because only one of the three directions would have this problem. Since $\lg n < \frac{\sqrt[3]{n}}{3}$ only for $n > 64000$, the universal computer would only excel on rather large problem instances.

It should probably be pointed out that circular reasoning was used in the above argument. We selected a simple form of enumerated expression (for other reasons) and observed that the parallel structures that result can be simply laid out in Euclidean spaces of various dimension. This may make it necessary to provide mechanisms to included "canned" subnetworks (i.e., for sorting), but once these are provided the system will be reasonably general and will retain the property that it generates easy-to-lay-out networks.

We explore a series of problems from the literature of computer science that are known to have good parallel solutions. We used classical problems from the literature of computer science, rather than, in any sense, selecting a "random cross section" of problems (whatever that would mean), because it is fairly well known what is possible in terms of parallel structures for these problems and we therefore had targets for the tools we were trying to build, as well as a yardstick against which to measure the results. We conjecture that most real problems that take a lot of computer resources reduce to a series of classical problems. For example, in [**Knu69**] and [**Knu73**] respectively the point is made that array manipulation and sorting are major consumers of computing resources as they are used today.

1.2.1 Previous Work

A technique similar to our virtualization technique is described in [Mir83] and [MWi84]. Their technique is to duplicate all scalers or array elements that receive multiple assignments and then to compute the data flow based on stereotyped constructs of a FORTRAN-like specification language. Their system finds interconnection nets that meet certain linear algebraic properties.

The main differences between the techniques in the above paragraph and our virtualization is in the form of data dependency allowed. The use of linear algebra for dependency analysis allows application of these techniques only when information flows from an array element to another array element whose coordinates are linear functions of the coordinates of the first element. While the prototype TRANSCONS uses linear algebra in place of a more general theorem prover and therefore shares this property, the form of the rule and the compartmentalization of the information makes addition of new knowledge simple.

One consequence of this is that there is no notion of aggregation. Because of the finality of the result it can not be aggregated conveniently, and it is therefore a reasonable technique only where there is a constant amont of work per processor element already.

Numerous systems exist for creating VLSI layouts or VLSI topological descriptions from low level description languages. In each case we will only cite one or two examples, with no intent to imply anything about those that are chosen on the one hand or left out on the other.

MACPITTS ([SSC82]), from the MIT Lincoln Laboratory, can produce VLSI layouts from a LISP-like language which includes constructs like (SETQ ...) to create a signal, (+ ...) etc. to specify arithmetic or logical operations on signals, and looping constructs.

MACPITTS determines and places the minimum number of functional modules to "execute" a given "program", creates a programmed logic array (PLA) to control these modules, and lays out the wires among these parts.

The Palladio system ([HTF83]) allows a user to interactively create a VLSI topology by "discussing" with the system what is to be done.

These VLSI design systems have as their primary goal the avoidance of the electronics pitfalls such as capacitance problems or delays on long lines and violations of the "design rules" of the technology that would make fabrication unreliable.

The communication of actors or closures between processors to model communication of problem data between processors has been current since [AHe77] and [The82]. Here actors are separate objects that do their work by sending and receiving messages. Such a transmission is an event. A message can be an actor. Conceptually the actors have independent existence, but of course they must have some physical realization and, assuming machines capable of processing messages by and for actors contained therein are called processors, the passing of messages among actors in different processors can model communication among the processors. In this work the actor is an object that entitles its holder to perform some action by invoking it, and it can be passed from one process to another. The receiving process can invoke it, and the work is performed in an environment derived from the processor that created the actor. In this thesis we will use *closures*, which are objects similar to actors that can only be invoked once and then cease to exist.

Divide & conquer is a powerful synthesis technique for efficient sequential programs (see [Smi83a] and [Smi83b]), but it has not been widely used to synthesize tree-structured collections of communicating processors, even though such an application would seem obvious because of the correspondence between the division process of divide & conquer and the branching structure of the desired collection of processors. The reason for this is that the synthesis process encounters technical problems when one tries to perform such a synthesis

in the obvious manner. The use of closures, similar to actors, mitigates these technical problems at the expense of requiring a more general theorem prover than is required to perform divide & conquer syntheses of sequential programs.

1.3 Approach

TRANSCONS specializes in two areas. As the first specialty it can synthesize crystalline networks of processors, in which members of a family of processors can be described by vectors of indices (integers initially; in principle any ordered set). In such a structure, each processor is connected to those other processors, each at a fixed distance and direction from this processor, that exist (if we visualize the network as a group of processors, each occupying a point with integer coordinates in Euclidean space of appropriate dimension). As the other specialty it can synthesize balanced binary tree structures in which the internal nodes all run the same procedure.

In all cases the synthesis process starts with specifications in the V language (see [Gre81], [GCP81], and [Kes85]). V is a broad-spectrum language based on first order logic (FOL) but containing locutions ranging from FOL to LISP- or Pascal-like specifications of individual operations, data structures, and values. We use this language for several reasons:

- The language is a good one for specifying rules used to transform specifications as well as the specifications to be transformed. It shares with LISP and RAPTS [Pai82] the property that programs in the language are normally expressed as instances of the data structures such programs most easily manipulate.
- These specifications are similar to first order logic expressions. A theorem proving capability is essential, and much work has been done on the problems of automatic theorem proving in first order logic expressions.

- Use of a broad-spectrum language facilitates a stepwise refinement. If the source and target languages were distinct rather than being parts of a single language, the creation of target text from source text would need to be conceptually a single step. An intermediate form that could hold both source and target locutions would have to be provided unless the process actually was so simple that a single examination of any source object was sufficient.

We augment the V language cited above with several constructs designed to specify interconnected collections of similar processors.

1.3.1 Parallel Structure Refinement

We develop a series of models of the parallel computation process. In the highest level, most details are unspecified; in the intermediate level, the order, but not the timing, of various communications and computations is described; and finally in the two lowest levels, the notion of a clock is introduced. In the higher of these two levels, the time at which various operations can take place is determined algebraically. In the lower level, time *differences* between actions are computed. This can be used directly in a VLSI synthesis. A computation in which operands are available simultaneously and the result is needed one cycle later can, for example, be performed by combinatorial logic with a single "latch" connected to the output.

The last stage of the refinement is future work, but we argue that it will meld well with the rest of TRANSCONS, and that TRANSCONS will then be able to transform first order logic specifications into circuit descriptions lacking only device placement steps to be complete VLSI chip descriptions.

The series of models is such that a coherent parallel structure results from stopping the synthesis process at any level. Some levels can not be reached by some specifications, and

the lowest levels may contain more detail than is desired. The user can control the extent of the synthesis process.

1.3.2 Crystalline Methods

Crystalline structure synthesis begins with transformations embodying data flow analysis and analysis of expressions comprising indices of references to array elements. Each intermediate datum in an array of the specification is assigned to a processor whose index corresponds to the index of the datum. The results of this simple analysis is generally a clumsy structure in which each processor is connected to many other processors – too many to be practical. Often there are other weaknesses in the structures. Additional techniques are therefore necessary to produce usable parallel structures, and TRANSCONS contains rules that embody these.

The first and most important of these techniques is *communication reduction*. In this technique, a rule seeks a set of communication lines whose transitive closure is equal to (or includes) a distinguishable subset of a given communication network. After replacing that portion of the network with the smaller set, we repeat the process.

A second technique is *aggregation*, or the collection of many processors into one. This technique has two important uses; reducing the number of processors in a system when each has too little work to do, and gluing together simple networks to make more complex ones.

A third technique is *virtualization*, or the increase of the dimensionality of a data structure by explicating a loop that repeats assignments to an internal register. TRANSCONS normally uses virtualization together with aggregation, because the former creates numerous processors with little work for each of them to do, and the latter combines processors. Every virtualization has an inverse aggregation, but whenever a virtualization creates an array of processors with more than two dimensions there is more than one aggregation, and

TRANSCONS has the option of choosing a different one from the inverse of the original virtualization.

The fourth technique is *chain creation*. When an asymptotically large number of connections exists between an I/O processor and the working processors, and TRANSCONS wants to reduce this number, it tries to apply this technique. If the sets of values used in the working processors can be grouped properly, then the wiring can be rearranged so only a few distinguished working processors are connected to input processors, and the rest can receive information "second hand" from other working processors. Similarly, for output, information can be collected from several working processors connected in a chain and sent to the outside world via a few distinguished processors.

We restrict the forms of the expressions in the declarations describing the processors and their interconnections, because use of a theorem prover is required for all of these techniques and the restrictions make this much more feasible.

We argue the adequacy of TRANSCONS's techniques for crystalline synthesis by showing some syntheses of parallel structures for dynamic programming and two structures for multiplication of matrices.

1.3.3 Tree Methods

We use divide & conquer as the primary synthesis tool for the creation of tree parallel structures. This technique has a long history of creating efficient sequential programs from specifications. While it would appear that the synthesis of a tree structure by divide & conquer should be immediate because of the correspondence of subproblems and subtrees, there are issues that require resolution.

We therefore introduce the notion of passing a *closure*, or functional object, between processors. While this is not novel (see, for example, [**Hew76**]), our use of it is. We are

given a specification of I/O behavior of two arrays (one an input and one an output). We transform this into two specifications: I/O behavior mapping an input array into a functional object exhibiting certain behavior given the input array of the original specification, and a request that this functional object be applied. As we demonstrate, using the combination of divide & conquer and the computation of well-chosen closures, TRANSCONS is able to synthesize a variety of tree parallel structures which can solve problems ranging in complexity from census functions [LiV81] (which require no closures) to parts of a connected components computation, in which a graph's adjacency matrix is read in row by row, and the parallel structure "learns" what the sets of points are such that there exists a path from any node in a set to any other node in the same set.

1.3.4 Additional Techniques

As part of a demonstration of the power of our techniques, we synthesize three circuits for the addition of binary numbers. The use of different combinations of techniques produces circuits occupying three places in a spectrum of speed/cost tradeoffs.

The first order logic specification for binary addition has nested bounded quantifiers arranged in such a manner that the bounds of the inner quantifier depend on the bound variable of the outer one. Therefore, the parallel structure synthesized by the previous methods of this thesis has $O(n^2)$ boolean values to compute for addition of two n-bit numbers. To accomplish this in $O(\log n)$ time would require $O(n^2)$ processors. We use a series of axioms and theorems relating the max, min, $\wedge$, $\vee$, $\exists$ and $\forall$ operators. An example of a necessary theorem is $\forall l < x < u[P(x)] \equiv \max_{x<u}[\sim P(x)] \leq l$, which restates a universally quantified expression bounded by an integer subrange into a maximization. The axioms and theorems, the proofs, and their use are described in Chapter 7. The process requires a theorem prover general enough to accept the axioms and to either prove or accept the theorems relating these operators. We achieve a specification in which a

pair of nested quantifiers is replaced by an arithmetic comparison of two bounded max operations. We call the entire process *quantifier levelling.*

1.4 Organization

The next Chapter (after this introduction) gives formal descriptions of the four levels of synthesis detail that TRANSCONS will be capable of when it is complete. The third Chapter describes the abilities of TRANSCONS that facilitate crystalline synthesis.

Chapter four discusses the divide-and-conquer method for synthesizing treelike parallel structures, explores some of the problems that must be solved to make it work, introduces the notion of a closure to solve these problems, and introduces the language we use to describe resulting structures. Chapter five gives several examples of the synthesis of tree structures by these methods, and discusses in detail methods for removing the closures, which are a necessary *"scaffolding" for the synthesis process* but not intended for the final product.

The sixth Chapter shows a case in which use of mathematical identities makes TRANSCONS more powerful than it otherwise would have been. It lends credence to our conjecture that our synthesis tools will turn out to be more powerful than it would seem from the apparently specialized nature of problems we solve in Chapter s three and six.

The seventh and last Chapter explains the significance of our results and the future paths we expect this research to take. A successful pursuit of this future research will enhance TRANSCONS to an extent that it will be able to automatically synthesize most of the parallel structures that have been created by hand, plus structures of comparable difficulty that have not been created yet, either because the need for them has not yet arisen or because they are so specialized that the effort has not been deemed worthwhile.

The appendix contains four sections: sample dialogs with a complete TRANSCONS, formal proof of correctness of one of the most important rules, formal proofs of the identities used in Chapter six, and a description of the theorem proving requirements of the crystalline synthesis portion of TRANSCONS.

Chapter 2

Formal Descriptions

The target of TRANSCONS synthesis has four levels arranged in a hierarchy. These range from a high-level description of computation activity to a level of description that lacks only device placement to be suitable for VLSI implementation. TRANSCONS refines specifications from the higher levels of this hierarchy to the lower by adding specialized information.

A higher level differs from a lower level by requiring more capabilities in the implementing hardware. As an example, the highest level ("multiprogramming") assumes that the implementing hardware is able to store indefinite amounts of information and to process each piece of information using a separate virtual processor (usually called a "process"). The lowest level requires only that each processing element be able to compute some simple function of all inputs present at one clock cycle c and to present the answer(s) at its output(s) at a clock cycle $c + i$ where i is a constant integer dependent on the processing element. We provide coherent synthesis levels, rather than merely describing a process in which the specification becomes more and more refined but in which parts of the specification may be in intermediate states that are not meant to be used by any entity except continued TRANSCONS synthesis, for two reasons. The first is that it is not possible to reduce every specification to the lowest level, and we therefore need coherent intermediate

levels as targets for these specifications. The second is that we may have hardware available that can meet the requirements of the higher levels, so we may choose to stop even though it would be possible to proceed.

We progress from higher to lower levels by adding specializing information to the specifications. To illustrate the various levels of representation we will use the following specification:

$$\forall A \exists A' \left[\forall i \in \{1 \ldots n\} \left[a'_i = \sum_{j \in \{1 \ldots i\}} a_j \right] \right].$$

The actual reduction operator, here shown as addition, is unimportant; what is important is that it be implementable as combinatorial logic in VLSI. We intend to show implementations that might result from taking this specification through all four levels of TRANSCONS synthesis. The reduction operator must be implementable in VLSI because we intend to display an implementation of the specification which uses the reduction operation, among other things, as an atomic operation.

2.1 Multiprogramming

A parallel structure in this level consists of *procedures*, *processors*, and *pools*. A processor contains one or more processes, each of which contains, in turn, a *procedure* and a (possibly null) *index variable binding*.

A processor is described by a processors declaration, whose components are given in this tree diagram:

PROCESSORS (index variables) enumerators for index variables
| **HAS** array name (array index variables) enumerators for same
| **HEARS** processor family name (indices) enumerators
| | (**USES** array name (indices) enumerators)
| **TALKS** processor family name (indices) enumerators
| | (**SENDS** array name (indices) enumerators)
| **LINKS** processor family name (indices) p f name 2 (indices) enumers
| | (**PASSES** array name (indices) enumerators)

Any of the subclauses can have a condition attached to it which will specify that the subclause only applies to instances of the processor family or enclosing clause for which the condition is true. The condition is restricted to Presburger Arithmetic expressions whose free variables are variables that are bound further outward in the processors statement, or not bound at all. The theorem prover will assume nothing (beyond type information) about an unbound variable, which we will call a *superglobal* in the following. We use the phrase *(sub)clause instantiation* to describe the instance of any clause or subclause with specific values of the bound variables. A processors declaration is a type that is attached to a name, which becomes the name of a processor family.

There are consistency requirements. If processor *A* **HEARS** or **LINKS** from processor *B*, then processor *B* must **TALK** (to) or **LINK** to processor *A*. If a **HEARS** clause has a **USES** subclause, the **HEAR**d processor must either **PASS** or **SEND** that value within the corresponding **LINKS** or **TALKS** clause. Note that this imposes a condition on clause instantiations, not merely clauses.

A procedure contains one or more statements from the V language. These include references, assignments, reduction operations, other operations, enumeration descriptions and block structure. For every reference in the procedure of a process it must be true that the **PROCESSORS** statement for that process has either a **USES** clause or a **HAS** clause for that reference.

Each reference to a value that is only available from another processor is an abbreviation for a "guarded command" [Hoa78] whose guard is the availability of the datum and whose action is the retrieval of the datum to the point of invocation of the reference.

Each

HEARS/TALKS, **HEARS/LINKS**(to), **LINKS**(from)/**TALKS** and **LINKS**(from) /**LINKS**(to) pair denotes a *pool*. In addition there is a single pool in each processor for local memory. The consistency rules require that there will be corresponding subclauses in each of these clause pairs. This means that, for example, everything that is **SEN**t is **USE**d. It also requires that both ends of a link be present. Each of these subclause pairs represents a *name*. The variable name together with its indices is used as the name in the local memory pool. Each **SENDS** clause instantiation must match a **USES** clause instantiation, and this condition can only be met if the clauses themselves match.

There are relationships between this model and the data flow machine models. See, for example, [TAm83], although the mechanism of that model differs from the mechanism we use. With data flow machines, each operation is represented by an object (sometimes a word in a memory, sometimes a physical processor) which has a name and which gives the name of one or more operands. Its name can be used as an operand in other operations. With our model the code fragments for the processors correspond to the operators, the **USES** clauses to the operand names, and the **SENDS** clauses to the exported operands.

There are two ways that multiple processes in one processor can be specified. One is by declaring multiple procedures in one processor. The other way is with enumeration statements. These are of the form

```
(in processor P) :
∀v ∈ s
   procedure
end
```

where s is set-valued. (The fragment (in processor P) is the declaration that the following procedure runs in processor P.) There is no commitment to a specific order. In this form of multiprogramming, a process is created for each instantiation of the ∀ variable. The processes have identical procedures except for this instantiation.

A pool contains triples of the form ⟨name, index, value⟩. Several operations are defined on pools, and the accesses to the pools that appear in the procedures must be drawn from this set.

A *reference* has the form $GET(options, pool)$ or $GET(options, pool, name)$ or $GET(options, pool, name, index)$. *options* is a two-tuple of one of *destructive*, *mark* = ⟨*mark*⟩, and **nil**; and either *hang* or *test*. The semantics of this is that the pool is checked for the presence of any datum matching as much as we know about the name and index. We return the value if there is one and we either return **false** if there isn't and we were *test*ing, or we suspend progress of the process if we weren't. The first part of the option describes what we do next if the retrieval was successful. If we were *destructive*, we delete the item; if we were *mark*ing, we mark the item so that a subsequent retrieval with the same *mark* = ⟨*mark*⟩ will not succeed with this item, and if the first option was **nil** we do nothing (and another retrieval request might pull the same item).

A *store* is of the form $PUT(pool, name, index, value)$. This modifies the state of the world so that a $GET(options, pool)$ can equal $(index, value)$, $GET(options, pool, name) = value$, and $GET(options, pool, name, index) = value$.

Grouping of the processes within a processor into superprocesses, which are collections of processes that intercommunicate more than other pairs of processes within a processor, are provided. They are specified by collecting the processes within a group into a supposedly independent "processor", and then collecting the "processors" with an **AGGREGATION** declaration. The **AGGREGATION** declaration has all of the components of a **PROCESSORS** declaration, plus a possibly enumerated list of the processors it contains. There is a consistency rule that requires that each **AGGREGATION HAVE, HEAR** etc. everything that its components **HAVE, HEAR** etc.

All of the values described in this Section can be closures (see Chapter 5) as well as ordinary values.

A simple multiprogramming solution to our sample specification,

$$\forall A \exists A' \left[\forall i \in \{1 \ldots n\} \left[a'_i = \sum_{j \in \{1 \ldots i\}} a_j \right] \right]$$

would be described as:

A **istype INBOUND ARRAY**$(\{1 \ldots n\})$
Pa **istype PROCESSORS HAS** $A_i, i \in \{1 \ldots n\}$
    TALKS $Pb_j, j \in \{i+1 \ldots n\}$ **(SENDS** A_i**)**
A' **istype OUTBOUND ARRAY**$(\{1 \ldots n\})$
Pa' **istype PROCESSORS HAS** $A'_i, i \in \{1 \ldots n\}$
    HEARS Pb_i **(USES** B_i**)**
B **istype ARRAY**$(\{1 \ldots n\})$
Pb **istype PROCESSORS** $i, i \in \{1 \ldots n\}$ **HAS** B_i
    HEARS Pa **(USES** A_i**)**
    TALKS Pb'_i **(SENDS** B_i**)**
B' **istype ARRAY**$(\{1 \ldots n\})$
Pb' **istype PROCESSORS** $i, i \in \{1 \ldots n\}$ **HAS** B'_i
    HEARS Pb_i **(USES** B_i**)**
    if $1 < i < n$ **then**
        LINKS Pb'_{i-1}, Pb'_{i+1}
            (PASSES $B_j, j \in \{1 \ldots i-1\}$**)**
    if $1 < i$ **then**
        HEARS Pb'_{i-1} **(USES** $B_j, j \in \{1 \ldots i-1\}$**)**
    . . .

$(in\ Pb_i'):$
$temp \leftarrow 0$
$\forall j \in \{1 \ldots i\}$
  $temp \leftarrow temp + B_j$
  end $\forall$
$B_i' \leftarrow temp$

Some irrelevant detail has been ignored, but key points are that all enumerations are unordered, and that in B_i' there are i processes waiting to finish. The enumeration is an abbreviation for text that updates *temp* and keeps track of whether it is complete so the assignment $B_i' \leftarrow temp$ can be made.

2.2 Single Process per Processor

This model is similar to "Multiprogramming" except for three features.

- The memory pools in this model are ordered. There are potentially multiple pools per **HEARS**, etc. clause, as one must be provided for each **USES**, etc. clause. These pools are either stacks or push-down lists, and the enumerations within the **USES**, etc. clauses must be ordered (enumerating through a sequence rather than a set).

- The **SENDS** and corresponding **USES** clauses can either have instantiations in the same or the opposite order, making the communication channel a pipeline or a pushdown stack respectively. This applies to communications links (where the source and destination are different processors) and to memory pools (where they are the same).

- Only one process is allowed in each processor.

This is a lower-level model than "Multiprogramming" because of the lack of the requirement that the hardware processing elements be able to run multiple programs effectively simultaneously. As before, more information must be supplied, consisting of modifications to the program to explicitly test for the availability of required data, and declarations of pools as separate objects.

It is possible to transform a static collection of simultaneously running programs into a single program with the same effect, provided that none of the constituent programs enters a nonterminating loop that performs no access to any pool. One such set of transformations would supply a master control program which would have as coroutines copies of the procedures of each of the multiple processes to be simulated. Each access to a pool in one of the simulated processes must be preceded by a test to see whether there is something there, and if there isn't the "process" co-returns to the master control program. It is clear that this preserves correctness, and if we assume that the processor has enough power to do the work assigned to it, we will not see a situation in which some work does not get done because some process's program runs indefinitely, always finding work to do.

This transformation can only be performed if a constant number of processes are to be folded. Code that satisfies the previous model may have a process per pool datum. For this reason we must impose the restriction that unordered enumerations are not permitted. The pools must be turned either into queues or pushdown stacks.

A pool is an object whose type is **pool**. It has a **stack?** property and it can have (or lack) a name. A pool is associated with every **USES**, etc. clause. Every pool is shared by one **SENDS** or **PASSES** clause as a source and one **USES** or **PASSES** clause as a sink[1]. The pool has **source** and **sink** properties, and the **USES**, etc. clauses have **pool** properties.

[1]It is possible for different instantiations of a single **PASSES** clause to be both the source and the sink.

Only a single process per processor can exist. The procedure may contain enumerations, but they must be ordered and they denote sequential composition and not parallel composition of tasks.

On this level it is important how many uses are made of a given datum, because the logical connection between a datum and its use is made by counting. For this reason some streams are duplicated, i.e., information is put in at one stream and removed at several. In this case, language like

. . .
(**USES** a...)
. . .
(**SENDS** b...)
. . .
(**SENDS** c...)
. . .

(in Px) :
$b, c \leftarrow a$

will be used to describe the copying of the stream that supplies a into ones that supply b and c.

Part of a parallel structure that satisfies the benchmark specification on this level follows:

A **istype INBOUND ARRAY**($[1 \ldots n]$)
Pa **istype PROCESSORS HAS** $A_i, i \in [1 \ldots n]$
TALKS $Pb_j, j \in [i+1 \ldots n]$ **(SENDS** A_i**)**
. . .
B' **istype ARRAY**($[1 \ldots n]$)
Pb' **istype PROCESSORS** $i, i \in [1 \ldots n]$ **HAS** B'_i
HEARS Pb_i **(USES** B_i**)**
if $1 < i < n$ **then**
LINKS Pb'_{i-1}, Pb'_{i+1}
(PASSES $B_j, j \in [i-1 \ldots 1]$**)**
if $i < n$ **then**
LINKS Pb_i, Pb'_{i+1}
(PASSES B_i**)**
if $1 < i$ **then**
HEARS Pb'_{i-1} **(USES** $B_j, j \in [i-1 \ldots 1]$**)**
. . .
$(in\ Pb'_i)$:
$temp \leftarrow 0$
for $j \in [i-1 \ldots 1]$
$temp \leftarrow temp + B_j$
end for
$B'_i \leftarrow temp$

The difference between this solution and the previous one is that all orderings are explicit. Note that the j enumerations are "backward". The **for** operator replaces the $\forall$ operator of the previous parallel structure description. Instead of specifying unordered and therefore potentially concurrent executions in one processor, it specifies "ordinary" looping.

2.3 Clocked

In the clocked level there is an object called a *clock*. It is permitted to take values from an ordered domain (conceptually, vectors of integers; usually a scalar of type integer). The print prototypes (syntax declarations) of TRANSCONS allow every **USES**, etc. clause to have an **AT** clause. Consistency rules require that effect follow cause. For example, no **SENDS** clause instantiation occurs **AT** any time before the computation it depends on

USES all of its values, plus an amount that depends on the nature of the computation as described below.

The additional specializing information beyond "Single Process" is the **AT** information. The value of the **AT** property of a **USES**, etc. clause is a parametric expression in variables bound in the scope of the clause. This allows the hardware of a processor to be simplified in several ways, depending on some of the values of the **AT** clauses.

The programs can be rewritten to not test whether data is available. Data is assumed to be available at the appropriate times, and the hardware can merely "gate in" data, or read the port without regard to signals describing the presence or absence of data. Similarly, data can be written without regard as to whether there is room for it.

All information present in "Single Process" is present for this level also. A few additional elements are added. There is a mapping $T : s \rightarrow i$ where s is an element of V syntax (a node type) and i is an integer. $T(s)$ will be called the ***intrinsic delay*** of type s. The interpretation of this is that if s is an operator (e.g., $+$) then $T(s)$ is the time required to perform the operation, and if s is atomic then $T(s)$ is the time required to develop the value. The time to evaluate $a + b$, for example, is $2T(\langle\text{variable reference}\rangle) + T(+)$.

$T(m)$ where m is a reduction operation is the time for a single step. This means that a single value or set of values will necessarily be absorbed after i time units.

A "Single Process" specification can be converted into a "Clocked" specification by addition of a clock declaration, and assigning times to each of the **USES** clauses. The transformation rules assign a time of 0 to the first instantiation of each **SENDS** clause, and use propagation techniques to assign times to other events. The **AT** expression of an instantiation of an output of a node must at least equal the highest of the sums of the **AT** values of the node's inputs and its intrinsic delay.

The implementation of our specification to this level is:

(Here we focus only on the actions of the Pb' processors, which are the ones that do the actual computation)

C **istype CLOCK** $[1 \ldots 3n]$

B' **istype ARRAY**$([1 \ldots n])$
Pb' **istype PROCESSORS** $i, i \in [1 \ldots n]$ **HAS** B'_i
HEARS Pb_i **(USES** B_i **AT** $C = 0$**)**
if $1 < i < n$ **then**
LINKS Pb'_{i-1}, Pb'_{i+1}
(PASSES $B_j, j \in [i-1 \ldots 1]$ **AT** $C = 3(i-j)+2$**)**
if $i < n$ **then**
LINKS Pb_i, Pb'_{i+1}
(PASSES B_i **AT** $C = 2$**)**
if $1 < i$ **then**
HEARS Pb'_{i-1} **(USES** $B_j, j \in [i-1 \ldots 1]$ **AT** $C = 3(i-j)$**)**

(in Pb'_i*) :*
$temp \leftarrow 0$ **AT** $C = 1$
for $j \in [i-1 \ldots 1]$
$temp \leftarrow temp + B_j$ **AT** $3(i-j)+1$
end for
$B'_i \leftarrow temp$

There is an invisible notation on the nodes whose printed representations are "$temp \leftarrow temp + B_j$" and $temp \leftarrow 0$ giving them **AT** properties. We have depicted the $temp \leftarrow temp + B_j$ and $temp \leftarrow 0$ lines as having visible **AT** clauses for clarity.

2.4 Fixed Delay Level

A possible endpoint of a synthesis is a structure amenable to VLSI implementation. In order for this to be feasible, several conditions must be met that aren't necessary for a parallel structure in which many Van Neumann computers cooperate. An example of this is the fact that the memory buffering a link between two computing elements must have a definite length, specified when the circuit is burnt.

The restrictions we must impose, that things happen at definite times and are separated by definite intervals, can best be modeled by providing declarations that the times things happen (**AT** clauses) are relative to the times other things happen. If the time difference is equal to an explicit constant, a VLSI synthesis system could use a shift register to control the timing and to model the data path. If the time difference is a superglobal this is still possible, although the circuit cannot actually be sized until the value of the superglobal is known.

At this level, the internal nodes of the processors' computation nodes as well as the **USES**, etc. clauses, except for the **SENDS** clauses of the input processors' **TALKS** clauses, have **AT** properties. The values of these properties are sets of pairs of other nodes and strictly positive integer constants instead of an expression. For each node n there is one **AT** property for each node m from which n receives data flow. The property will be the pair of m and a positive integer i, and the semantics of this is that n finishes its computation and makes available its output values i units of time (clock cycles) after m does. If there is a list $(m, m_1, m_2, \ldots, m_k, n)$ such that $(m_k, i_k) \in (\mathbf{AT}\ n)$, $(m_j, i_j \in (\mathbf{AT} m_{j+1})$, ..., $(m, i_0) \in (\mathbf{AT})$ then there is said to be an **AT**-path from m to n with delay $\sum_{0 \le j \le k} i_j$. The graph whose nodes are the nodes of the specification and whose edges are the **AT**-links is a DAG, but it is not necessarily a tree. Two paths from m to n must have the same delay. The delay of each node must be greater than or equal to the intrinsic delay of the node's type.

The motivation for this level of description is that a specification that meets these conditions can be simply transformed into suitable input for a VLSI placement program by taking several steps, assuming that all nodes of the specification have types whose operator can be implemented as a single object in VLSI. If this is not the case (for example if the specification includes a multiplication node and the library has only addition), the offending node must first be broken down into simpler nodes. The intrinsic delay of each

node type should equal the number of clock cycles required for the circuit that implements the function to work. If $(m,i) \in (\mathbf{AT}n)$ and the intrinsic delay of n is $j = i - k$, then fabricate a wire from the circuit implementing m to a shift register with k elements whose output is connected to the appropriate input port of n.

It is more difficult to display the implementation of our specification on this level, because the **AT** clauses do not equate C with a form whose free variables are indices but instead with an object with two slots; another node and an integer constant. The structure that results is circular. We will display this by giving names of the form ⟨script alphabetic:⟩ to some nodes so they can be referred to by **AT** clauses. Again we will depict the $temp \leftarrow temp + B_j$ and $temp \leftarrow 0$ lines as having visible **AT** clauses.

C **istype CLOCK** $[1 \ldots 3n]$

B' **istype ARRAY**$([1 \ldots n])$

Pb' **istype PROCESSORS** $i, i \in [1 \ldots n]$ **HAS** B'_i
HEARS Pb_i $(\mathcal{A}$: **USES** B_i **AT** $C = 0)$
if $1 < i < n$ **then**
LINKS Pb'_{i-1}, Pb'_{i+1}
$(\mathcal{B}$: **PASSES** $B_j, j \in [i-1 \ldots 1]$ **AT** $\mathcal{E}_{j-1} + 1)$
if $i < n$ **then**
LINKS Pb_i, Pb'_{i+1}
$(\mathcal{C}$: **PASSES** B_i **AT** $\mathcal{F} + 1)$
if $1 < i$ **then**
HEARS Pb'_{i-1} $(\mathcal{D}$: **USES** $B_j, j \in [i-1 \ldots 1]$ **AT** $B_j + 1)$

$(in\ Pb'_i)$:
$\mathcal{E}$: $temp \leftarrow 0$ **AT** $\mathcal{A} + 1$
for $j \in [i-1 \ldots 1]$
$\mathcal{F}$: $temp \leftarrow temp + B_j$ **AT** $\mathcal{D}_j + 1$
end for
$B'_i \leftarrow temp$

2.5 Summary

We have seen that there are four useful description levels for parallel structures.

The highest level, *multiprogramming*, is useful when a processor that can efficiently perform context switching and can contain memory pools would not be objectionable. This would be the case when a typical microprocessor with sufficient memory could be provided for each processor.

The second level, *single process*, is useful when a general purpose processor, but no context switching or memory pool management mechanism, can be provided. An example of such a situation would be the use of a typical microprocessor with no memory in addition to its internal memory. This internal memory is normally insufficient to hold several inactive contexts.

The third level, *clocked*, covers situations in which a processor simple enough to do things in a specific, fixed manner and order is desirable, but enough logic and memory can be provided to allow for the storage and retrieval of some intermediate values. An example of a technology that would be appropriate is interconnections of finite state machines and FIFO and LIFO devices.

The lowest level, *fixed delay*, must be reached when the computation elements are restricted to combinatorial logic and latches. Only fixed time differences between the occurrences of various events are allowed.

We therefore have a series of synthesis levels and corresponding computation models for the four main technologies in which one would want to implement a parallel structure.

Chapter 3

Case Studies of TRANSCONS Techniques

To develop the techniques described above, we have explored efficient parallel structures for several classical problems and algorithms, described in the following Sections. In all cases there will be a series of specifications, separated by rules and prose, describing a series of states of a node currently being transformed by TRANSCONS. We will highlight the changes with a vertical stroke (|) in the left margin, but will supply the entire current state of the node for reference. A node in V roughly corresponds to a syntactic object in a syntax tree.

3.1 Polynomial-Time Dynamic Programming

We have examined a class of polynomial time (P-time) dynamic programming algorithms for which it is possible to synthesize an optimal parallel scheme. The synthesis uses rules displayed below, and inference capabilities described in [Bro82]. Abstractly programmed algorithms in this class include the Cocke-Younger-Kasami parsing algorithm for a fixed, possibly ambiguous Chomsky Normal Form grammar, described in [AUl72]; the Optimal Binary Search Tree algorithm, described in [Knu73]; and Optimal Multiple Matrix Multiplication, described in [AHU74]. All of the algorithms fit into the following scheme.

Each algorithm generates the "solution" to a problem whose input is a sequence S of n items by using a dynamic programming technique. This technique generates a solution for a sequence of items by combining solutions for contiguous subsequences. The solution $V(K)$ for a sequence K of length n is found by:

1. Generating the $n-1$ possible partitions of K into contiguous nonempty subsequences I and J such that $I||J = K$;
2. Forming for each partition a partial solution for $I||J$ by applying a function F to $V(I)$ and $V(J)$;
3. Obtaining $V(I||J)$ by combining (using a binary operation $\odot$) all of the partial solutions. This is expressed formally below:

$$V(K) = \bigodot_{I,J:I||J=K} F(V(I),V(J))$$

In order to obtain the following parallel structure and have it run in time $\Theta(n)$, two conditions must hold:

- Both $\odot(x,y)$ and $F(x,y)$ must take constant time,
- $\odot$ must be associative. This allows $F(V(I),V(J))$ values to be included in the running $\odot$-total in any order they become available.

These conditions are met by a sizable class of problems, e.g., the problems mentioned above. The dynamic programming scheme described above generates the solution $V(S)$ for the original problem S of length n. The process starts with $V((s_i))$ for each $s_i \in S$, then generates solutions for subsequences of length 2, 3, and so on, up to n. We give below two dynamic programming algorithms that fit into this scheme.

The Cocke-Younger-Kasami algorithm parses a sequence of terminal symbols according to a fixed context free grammar in Chomsky Normal Form. This form specifies that each production rule in the grammar is either of the form $N \rightarrow t$ for some nonterminal N and terminal t, or $N \rightarrow PQ$ for nonterminals N, P, and Q. In this parsing algorithm, each problem is a sequence of terminal symbols, T, and the solution $V(T)$ is the set of nonterminal symbols that derive T. Let the initial terminal sequence be $(t_1 \ldots t_n)$. Then $V((t_i))$ are those nonterminals N for which there is a production rule in the grammar of the form $N \rightarrow t_i$. Given two contiguous sequences of terminals A and B, the nonterminals that produce $A||B$ include those nonterminals N for which there is a rule $N \rightarrow PQ$ where $P \in V(A)$ and $Q \in V(B)$. The nonterminals that produce a sequence S are obtained by dividing the sequence S into two subsequences in all possible ways and taking the union of the results. In our formalism,

$$F(V(S),V(T)) = \{N|[N \rightarrow PQ] \in G \wedge P \in V(S) \wedge Q \in V(T)\}$$

and

$\odot$ is the union operation, which is indeed associative.

Another example of a dynamic programming algorithm fitting our scheme is finding the complexity of the optimal grouping to multiply a given sequence $(M_1, M_2, \ldots, M_n)$ of matrices. Since matrix multiplication is associative, multiplying the matrices in different groupings produces the same result matrix, but different groupings may have different

execution efficiencies. If M is a $p \times q$ matrix, and N is a $q \times r$ matrix, then the product $M \times N$ will be a $p \times r$ matrix, and the multiplication will execute in time proportional to pqr (if a simple matrix multiplication algorithm is used).

This problem fits into the scheme presented above in the following fashion. The "solution" for each matrix subsequence $V((M_i, \ldots, M_j))$ is a triple (p, q, c): p is the row size of M_i; q the column size of M_j (since multiplication using any grouping of $(M_i, \ldots, M_j)$ results in a $p \times q$ matrix) and c is the optimal execution cost for computing $M_i \times \cdots \times M_j$. The F for this algorithm is defined below:

$$F((p_1, q_1, c_1), (p_2, q_2, c_2)) \equiv (p_1, q_2, c_1 + c_2 + p_1 q_1 q_2)$$

$\odot$ for this algorithm returns the triple with the minimum cost element. (Since only the costs can differ among triples, $\odot$'s choice is arbitrary if the costs happen to be the same.) The *minimum* operation is associative and commutative.

A high-level specification of the dynamic programming algorithm is presented below. A subsequence can be represented by its length and where it begins. The array A used below contains solutions to subsequences: the element $A_{l,m}$ contains $V((i_l, \ldots, i_{l+m-1}))$, where I is the initial sequence. The complexity of each "executable" statement is presented at the right.

The algorithm specification is as follows:

A istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le l \le n-l+1$
v istype **INPUT ARRAY** $(l), 1 \le l \le n$

$\forall\, l \in (1 \ldots n)$	$\Theta(1)$
$\quad A_{l,1} = v_l$	$\Theta(n)$
$\forall\, m \in (2 \ldots n)$	$\Theta(1)$
$\quad \forall\, l \in \{1 \ldots n-m+1\}$	$\Theta(n)$
$\quad\quad A_{l,m} = \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$	$\Theta(n^3)$

Figure 3.1: Specification of $\Theta(n^3)$ Dynamic Programming

F and $\odot$ because it is given that a single evaluation of both F and $\odot$ takes constant time.

The time complexity of the specified algorithm is indeed $\Theta(n^3)$ when executed on a sequential machine. A trick is available for one of the problems, Optimal Binary Search Tree of [Knu73]. This trick involves bounding k in Figure 3.1 more narrowly than $\{1 \ldots m-1\}$. This trick reduces the algorithm's running time to $\Theta(n^2)$, but it does not generalize to the other algorithms.

It is possible to implement the specification on a two-dimensional array of $\Theta(n^2)$ processors and the resulting structure will solve n-element problem instances in $\Theta(n)$ time. We know of no analog to the trick mentioned above for parallel structures. The memory size of each processor is $\Theta(n)$. Below we describe the operation of the structure, and then prove that it is a $\Theta(n)$ algorithm. This parallel structure has been reported in the literature [GKT79].

The network of processors is displayed in Figure 3.2. Observe that $P_{l,m}$ is connected to $P_{l,m-1}$ and $P_{l+1,m-1}$. Each processor $P_{l,m}$ will compute the value of $A_{l,m}$. To do this it needs two streams of information: $A_{l,k}$ and $A_{l+k,m-k}$, where $k < m$. These streams of data come respectively over wires from processors $P_{l,m-1}$ and $P_{l+1,m-1}$. Each processor $P_{l,m}$ (except $P_{1,n}$) will send every A-value received from $P_{l,m-1}$ to $P_{l,m+1}$ and from $P_{l+1,m-1}$

to $P_{l-1,m+1}$ as soon as $P_{l,m}$ gets it. Each processor will also compute F-values and merge them into a running $\odot$-total as soon as it gets the necessary A-values.

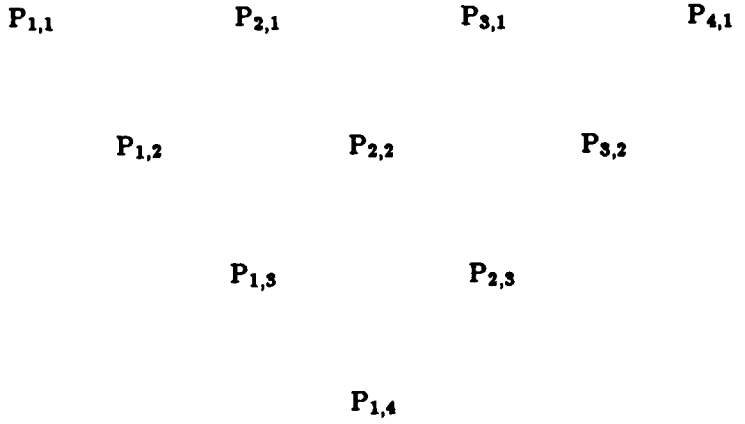


Figure 3.2: Processor Interconnections

At first glance, it might appear that this algorithm has time complexity $\Theta(n^2)$. Each processor needs to receive $\Theta(n)$ A-values from each of its incoming wires; it must at some time perform $\Theta(n)$ worth of computation on the data received before it sends its result on each of its outgoing wires. However, a careful timing argument shows that an execution time of $\Theta(n)$ can be achieved.

Definition 3.1 *Within* $P_{l,m}$, *for any* k *where* $1 \leq k < m$, $A_{l,k}$ *and* $A_{l+k,m-k}$ *are called a* complementary pair *of A-values.*

Processor $P_{l,m}$ will apply F to each complementary pair of A-values.

The next lemma shows that each processor $P_{l,m}$ receives all $2m-2$ values it needs, though it waits $\Theta(m)$ for its first complementary pair, $A_{l,\lceil m/2 \rceil}$ and $A_{l+\lceil m/2 \rceil, m-\lceil m/2 \rceil}$.

Lemma 3.1 *Each processor* $P_{l,m}$ *where* $1 \leq m \leq n-m+1$ *receives the values* $A_{l,m'}$ *where* $1 \leq m' < m$ *and (separately)* $A_{l+m-m',m'}$ *where* $1 \leq m' < m$, *in order of increasing* m'.

Proof: By induction on m. Clearly this is true for $P_{l,2}$, which receives only one value on each of its incoming wires. Now suppose it is true for $P_{l,m-1}$ and $P_{l+1,m-1}$. Then $P_{l,m}$ will receive A-values in the proper order from $P_{l,m-1}$ and $P_{l+1,m-1}$ through $m' = m-2$, following which it receives $A_{l,m-1}$ and $A_{l+1,m-1}$ from those processors. But the latter two A-elements are just those required to preserve the sequences. ∎

Let T be a time-dependent variable such that at system startup $T = 0$, and after x units of time $T = x$. The time unit satisfies the first condition of the following lemma.

Lemma 3.2 *If all of the following conditions are met:*

- All of the following takes processor $P_{l,m}$ no more than one unit of time: receiving two values, one each from $P_{l,m-1}$ and $P_{l+1,m-1}$; sending these values on to $P_{l,m+1}$ and $P_{l-1,m+1}$; applying the function F twice to two complementary pairs of A-values if all values are available; and merging the resulting value into a running $\odot$-total.
- The A-values come into $P_{l,m}$ in the order indicated by Lemma 1.2.
- Each processor $P_{l,m}$ sends values received from $P_{l,m-1}$ resp. $P_{l+1,m-1}$ to $P_{l,m+1}$ resp. $P_{l-1,m+1}$ no later than one time unit after receipt.
- At $T = 0$ processor $P_{l,1}$ transmits $A_{l,1}$.

then $P_{l,m}$ will compute $A_{l,m}$ no later than $T = 2m$.

Proof: By induction: $P_{l,1}$ is initialized to know $A_{l,1}$. Now suppose the lemma is true for $m < i$ and we wish to show it for $m = i$. We first show the following claim: that at $T = m+j$ $P_{l,m}$ will have included at least $\max(0, 2(j - \lceil m/over2 \rceil))$ F-values in its running $\odot$-total. This claim is proven by induction on j. When reading the proof of the claim, keep in mind that the "life" of a processor $P_{l,m}$ is divided into three epochs:

1. When $T < m$, the processor may have received no A-values.

2. When $m \leq T \leq \frac{3}{2}m$, the processor will have received at least $T - m$ A-values from each of its input lines. Since the first half of the A-values from each inbound wire form complementary pairs with the *last* half of the values from the other inbound wire, $P_{l,m}$ may not have been able to perform any calculations of any F-values yet.

3. When $T > \frac{3}{2}m$, the processor will have received at least half (more accurately, at least $m-T$) of the values from each inbound wire. During each unit interval, it will receive one A-value from each inbound wire, which will form a complementary pair with some value that was stored from the other wire during epoch 2. Two F-calculations will be possible – one pairing each of the just-received inbound data with a previously received input datum from the other side (unless m is odd and $T = m + \frac{m+1}{2}$, in which case the two values arriving at this time form a complementary pair).

If $j = 0$ the claim requires nothing. If $j > 0$, consider the situation at $T = 2(i - b)$. All processors $P_{l,k}$ and $P_{l+k,k}$, where $k \leq i - b$ will have completed their work. Their answers will have had time to reach $P_{l,i}$ after b time units, or at time $T = 2i - b$. But $j = i - b$, so by $T = i + j$ at least $2j$ A-values will have arrived from each input connection, and since there only i complementary pairs if $j > \frac{i}{2}$ only $2(i - j)$ pairs can be incomplete, meaning that at least $2(j - \lceil \frac{i}{2} \rceil)$ pairs are complete. Since, by induction on j, two time units ago

$2(j - \lceil \frac{i}{2} \rceil) - 2$ F-values had already been merged into the running $\odot$-total there is plenty of time to merge two new F-values into the running $\odot$-total, completing the induction step of the claim.

Lemma 3.2 follows immediately from the claim and the observation that the merging of $m-1$ F-values into the running $\odot$-total in $P_{l,m}$ constitutes a calculation of $A_{l,m}$.

Theorem 3.3 *The time to compute $A_{1,n}$ is $\Theta(n)$.*

Proof: Immediate from Lemma 3.2 ∎

A similar but more general result will be shown in Appendix Section B. We will show how this parallel structure can be derived from the specification in Figure 3.1.

3.1.1 Preparatory Rules

The problems amenable to TRANSCONS synthesis have internal arrays of storage, and the requirement must be to fill in an array by computing a value for each element. Our strategy will be to assign a processor to each element of the array. The first preparatory rules, *MAKE-PSS* and *MAKE-IOPSS* declare a processor family for each array of the probl and compose a single enumerated **PROCESSORS** declaration. This declaration has several clauses: the processors definition clause, the **HAS** clause, the **HEARS** clause(s), and the **USES** clause(s). **PROCESSORS** declarations were described in Section 2.1, but we will give a more complete example below. Any part of the **PROCESSORS** declaration except the processors definition clause can be made conditional.

P **istype PROCESSORS** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$
HAS $A_{l,m}$
if $m = 1$ **then HEARS** Q **(USES** v_l**)**
if $2 \leq m \leq n$ **then**
HEARS $P_{l,m-1}$ **(USES** $A_{l,k}, 1 \leq k \leq m-1$**)**
HEARS $P_{l+1,m-1}$ **(USES** $A_{l+k,m-k}, 1 \leq k \leq m-1$**)**
if $1 \leq m \leq n-1$ **then**
TALKS $P_{l,m+1}$ **(SENDS** $A_{l,k}, 1 \leq k \leq m+1$**)**
if $1 \leq m \leq n-1 \wedge l \geq 2$ **then**
TALKS $P_{l-1,m+1}$ **(SENDS** $A_{l-k,m+k}, 1 \leq k \leq m+1$**)**
if $1 \leq m \leq n-1 \wedge 1 \leq m \leq n-1$ **then**
LINKS $P_{l,m-1}, P_{l,m+1}$
(PASSES $A_{l,k}, 1 \leq k \leq m-1$**)**
if $1 \leq m \leq n-1 \wedge 1 \leq m \leq n-1 \wedge l \geq 2$ **then**
LINKS $P_{l+1,m-1}, P_{l-1,m+1}$
(PASSES $A_{l-k,m+k}, 1 \leq k \leq m-1$**)**

This declaration means all of the following:

- A family of processors exists. The family name is P. Each member of the family is named by two indices, and any member $P_{l,m}$ exists if $1 \leq m \leq n \wedge 1 \leq l \leq n-m+1$. The value n is an externally defined constant value (for any instance of the problem) defining the problem size. This **PROCESSORS** declaration actually declares some facts about every processor in the family.

- Each element, $P_{l,m}$, of this family is responsible for computing the value of (i.e., **HAS**) $A_{l,m}$. A is an array declared elsewhere in the specification that contains the **PROCESSORS** declaration.

- If $P_{l,1}$ is defined it needs v_l to compute its **HAS** values, and it expects to get these values from (i.e., **HEARS**) the (only) processor in the Q family.

- If $P_{l,m}$ is defined and $2 \leq m \leq n$, then $P_{l,m}$ needs the values of $A_{l,k}$ for any k, $1 \leq k \leq m-1$. It also needs $A_{l+k,m-k}$ for any k in that range. It expects to get these

values from processors in the P family, namely $P_{l,m-1}$ and $P_{l+1,m-1}$. The scope of the bound variables list (in this case, "l, m") is the entire **PROCESSORS** declaration.

- Similarly, processors whose indices meet certain conditions **TALK** to other processors and **SEND** certain values as specified by the enumerated expressions. Processors are also declared as **LINK**ing pairs of other processors and **PASS**ing sets of values.

The **TALKS/SENDS** and **LINKS/PASSES** information is redundant; this information can be inferred from the **HEARS/USES** data. In what follows, I will omit this redundant information to enhance readability except where I judge it to be critical to an understanding of the declaration.

3.1.1.1 Rule *MAKE-PSS*: Give Each Non-I/O Array Element its Own Processor

By our conventions, the portion before the "→" is the *antecedent* and the rest is the *consequent*. Variables free in the antecedent are implicitly existentially quantified and the scope of this quantification is the entire rule. Variables free only in the consequent are universally quantified (but this is rare). A rule is said to *apply* if the antecedent is true; when this happens the semantics of the rule is to make the consequent true. It is explicitly permissible for the consequent to make the antecedent no longer true.

rule *MAKE–PSs* (**) **TRANSFORM**

$**$ = 'bind *NAME* istype *X*'
$\wedge X$ = '**ARRAY**$(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$'
$\wedge$ **undefined** (*IO X*)
$\wedge Y$ = (**gensym** 'PROC)
$\wedge Z$ = '*PROCESSORS* (*BOUND*) *ENUMERS* **HAS** $NAME_{BOUND}$'
$\rightarrow$
$**$ = 'bind ... *Y* istype *Z*'

MAKE–PSs applied to Figure 1 binds as follows:

bindings:

$**$= $\langle$(*entire specification*)$\rangle$
= '**ARRAY** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$'
NAME= '*A*'
BOUND= 'l,m'
ENUMERS= '$1 \leq m \leq n, 1 \leq l \leq n-m+1$'
Y= 'P'
Z= '**PROCESSORS** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$
HAS $A_{l,m}$'

obtaining

A **istype** **ARRAY** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$
P **istype** **PROCESSORS** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$ **HAS** $A_{l,m}$
v **istype** **INPUT ARRAY** $(l), 1 \leq l \leq n$
O **istype** **OUTPUT ARRAY**

$\forall\, l \in \langle\langle 1 \ldots n \rangle\rangle$	$\theta(1)$
$A_{l,1} = v_l$	$\theta(n)$
$\forall\, m \in \langle\langle 2 \ldots n \rangle\rangle$	$\theta(1)$
$\forall\, l \in \{1 \ldots n-m+1\}$	$\theta(n)$
$A_{l,m} = \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$	$\theta(n^3)$
$O = A_{1,n}$	$\theta(1)$

as the new state of the database.

3.1.1.2 Rule *MAKE–IOPSS*: Assign I/O Arrays to Processors

This rule assigns a *single* processor to each input or output array. The reason only a single processor is assigned is that it is assumed that input values will reside in a single entity, such as a tape drive.

rule *MAKE–PSs* (**) **TRANSFORM**
  ** = 'bind *NAME* istype *X*'
  ∧ *X*'**ARRAY**$(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$'
  ∧ **defined** (*IO X*)
  ∧ *Y* = (**gensym** 'PROC)
  ∧ *Z* = '*PROCESSORS* **HAS** $NAME_{BOUND}$'
→
  ** = 'bind ... *Y* istype *Z*'

Rules *MAKE–PSs* and *MAKE–IOPSs* make **PROCESSORS** declarations that do not have **USES** and **HEARS** clauses yet. The next rule fills in those clauses, and subsequent rules improve them.

Rule *MAKE–IOPSs* applies for two sets of bindings:

** = ⟨(*entire specification*)⟩	** = ⟨(*entire specification*)⟩
X = ' **OUTPUT ARRAY** *O*'	*X* = ' **INPUT ARRAY** $v_l, 1 \leq l \leq n$'
IO = ' **OUTPUT**	*IO* = ' **INPUT**
NAME = *O*	*NAME* = *v*
BOUND = ⟨*empty binding list*⟩	*BOUND* = '*l*'
ENUMERS = ⟨*empty binding list*⟩	*ENUMERS* = '$1 \leq l \leq n$'
Y = R	*Y* = Q
Z = ' **PROCESSORS** R **HAS** *O*'	*Z* = ' **PROCESSORS** Q **HAS** $v_l, 1 \leq l \leq n$'

resulting in

(*P*.1)
A **istype ARRAY** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$
P **istype PROCESSORS** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$ **HAS** $A_{l,m}$
v **istype INPUT ARRAY** $(l), 1 \leq l \leq n$
| Q **istype PROCESSORS HAS** $v_l, 1 \leq l \leq n$

	O istype **OUTPUT ARRAY**	
	R istype **PROCESSORS HAS** O	
	$\forall\, l \in (1 \ldots n)$	$\theta(1)$
$(P.1a)$	$\quad A_{l,1} = v_l$	$\theta(n)$
	$\forall\, m \in (2 \ldots n)$	$\theta(1)$
	$\quad \forall\, l \in \{1 \ldots n-m+1\}$	$\theta(n)$
$(P.1b)$	$\quad\quad A_{l,m} = \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$	$\theta(n^3)$
$(P.1c)$	$O = A_{1,n}$	$\theta(1)$

So far, all rule application can be done in a straightforward manner, without inference.

3.1.1.3 Rule *MAKE–USES–HEARS*: Determine Processors' Inputs

We need rules to describe the connections between processors and the data that processors need to produce results. This rule is very conservative – it determines what array values each processor P^* needs, and it specifies a direct connection from the processors holding those values to P^*. The **USES** clause describes the *values* that a processor needs; the **HEARS** clause describes the *processors* that have (**HAS**) these values.

To determine this, consider the innermost loop which assigns values to array elements indexed by non-region-constants. Note that the form of the rule shown below evidences a need for elaborate flow analysis. Non-constant array index expressions are used as processor indices. The indices for those array elements whose values can affect the assigned value comprise the index expressions for the **USES** and **HEARS** sets. A reference at the same loop level will normally generate **USES** and **HEARS** clauses with null enumerations. A reference contained in a deeper loop will normally generate instances of such clauses with inherited enumerators from the loops.

```
rule MAKE-USES-HEARS (**) TRANSFORM
    CB = 'bind' ** ...'
   ∧ ** =
'PDCL istype PROCESSORS (PBV) PENUMER HAS ANAME_BINDEX'
   ∧ X = (INNER-LOOP-THAT-DEFINES ANAME CB)
   ∧ Y ∈ (ARRAY-REFERENCES-AFFECTING X)
   ∧ Z = (EFFECTIVE-ENUMERATOR-OF Y X)
   ∧ W. CONDITIONS =
CB. CONDITIONS ⋃(INFERRED-CONDITIONS X)
   ∧ W. CLASS = USES-CLAUSE
   ∧ W. ARG = 'ANAME
              (REL-BV PBV X. DEF-OF . INDEX-EXPR Y Z)
                  (RELENUMER PBV X. DEF-OF . INDEX-EXPR Y Z)'
   ∧ Q. CONDITIONS =
CB. CONDITIONS ⋃(INFERRED-CONDITIONS X)
   ∧ Q. CLASS = HEARS-CLAUSE
   ∧ HISBV = ANAME. PROCSTMT . PROC-BV-OF
   ∧ Q. ARG = 'ANAME. PROC-OF
                  (REL-BV HISBV
                       X. DEF-OF . INDEX-EXPR Y Z)
                  (RELENUMER HISBV X. DEF-OF . INDEX-EXPR Y Z)'
 →
  W ∈ **.clauses
   ∧ Q ∈ **. clauses
```

The INNER-LOOP-THAT-DEFINES function finds an innermost locality where an element from the argument array is defined (not merely used). The ARRAY-REFERENCES-AFFECTING function returns a set of all points in the program where an array is referenced and the value returned can affect the results of its operand, a program point. The EFFECTIVE-ENUMERATOR-OF function determines what (possibly implicit) enumerators its first argument (an array reference) is controlled by, beyond the enumerators that control its second argument (an array definition in this case).

The map, x.CONDITIONS, allows any node x to be placed under the influence of conditions (an if clause). INFERRED-CONDITIONS is a function that produces an if

clause that specifies exactly those conditions that must be true for the point representing the argument to be reached (a form of assertion propagation).

REL-BV and **RELENUMER** give a piece of text that respectively will serve as a bound variable and an enumerator for the fragment enumerated by the fourth argument to be valid for the third argument in the context of the second argument, using the bound variables of the first argument. This would be the bound variables of the fourth argument unless there is a variable name clash.

This modifies the first **PROCESSORS** type declaration, which becomes

P istype **PROCESSORS** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$
HAS $A_{l,m}$
| **if** $m = 1$ **then HEARS** Q **(USES** v_l**)**

Application to the assignment to $A_{l,m}$ in $(\mathcal{P}.1b)$ produces

P istype **PROCESSORS** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+1$
HAS $A_{l,m}$
if $m = 1$ **then HEARS** Q **(USES** v_l**)**
| **if** $2 \leq m \leq n$ **then**
| **HEARS** $P_{l,k}, 1 \leq k \leq m$ **(USES** $A_{l,k}, 1 \leq k \leq m$**)**
| **HEARS** $P_{l+k,m-k}, 1 \leq k \leq m$ **(USES** $A_{l+k,m-k}, 1 \leq k \leq m$**)**

Finally, apply *MAKE-USES-HEARS* one last time, to the null "enumeration", $(\mathcal{P}.1c)$, that sends the output value to the output "array", O. This forces us to modify R's **PROCESSORS** declaration as follows:

R **lstype PROCESSORS HAS** O
| **HEARS** $P_{1,n}$ **(USES** $A_{1,n}$**)**

This declaration is in its final form.

The applications of *MAKE–USES–HEARS* require flow analysis and some ability to reason about enumeration (to construct if clauses).

3.1.2 Optimization Rules

The rest of the rules described in this section will transform the simplest parallel structures into more efficient ones. They do this by detecting and removing redundant interconnections.

3.1.2.1 Rule *REDUCE–HEARS*: Improve HEARS clauses

It may be that a **HEARS** clause of a **PROCESSORS** declaration requires each processor to be connected to an asymptotically growing number of other processors. This is undesirable, because the number of interconnections in the whole collection of processors would grow faster than the number of processors, and the cost of interconnections would exceed the cost of processors for sufficiently large problems. This would, in turn, decrease the size of the largest problem that could be handled by a given parallel structure.

However, often it is not necessary for each processor to be connected to all other processors whose values it needs. If processor P_a needs values from processors P_b and P_c, but P_b needs a value from processor P_c, it may not be necessary for P_a to be connected to P_c. P_a must be connected to P_b, but P_b will be able to get the value that P_a wants from P_c, so it (P_b) can pass that datum along.

This form of this observation only secures a constant factor reduction in the number of interconnections (in this case, from two to one), but it is possible to do better by extending

the principle. Suppose, for example, that a structure includes a family of processors P_i for $1 \leq i \leq n$. Further suppose that $\forall i, j$ where $j < i$, P_i needs values from P_j. In this case, P_{i+1} will need all the values P_i needs, plus the value in P_j itself.

Basic Observation 3.1 *If P_j is capable of supplying all of the information that P_{j+1} needs, so it is possible to modify the structure to replace the $\Theta(n)$ connections required by this* **HEARS** *clause by a single connection.*

Definition 3.2 *In a parallel structure, a* family *of processors is the set of processors defined by a single* **PROCESSORS** *declaration when enumerated over the* **PROCESSORS** *clause's enumerator. That family is* generated *by that* **PROCESSORS** *declaration.*

Definition 3.3 *The set of processors in a processor P_a's family* **HEAR***d by P_a due to a* **HEARS** *clause H_0 will be written $H_0(P_a)$, and is called the* induced set of H_0 at P_a.

Definition 3.4 *Consider $H_0(P_a)$ and $H_0(P_b)$. Suppose that each is a subset of the same family as P_a and P_b (which are in the same family because they both have the same* **HEARS** *clause, H_0). The interconnections defined by H_0* telescope *if these sets $H_0(P_a)$ and $H_0(P_b)$ either are disjoint or one strictly contains the other, for any choice of P_a and P_b in the family. We also say that H_0* telescopes. *If $\forall_{P_a, P_b \in family} : [\emptyset \subset H_0(P_a) \subset H_0(P_b) \Rightarrow \exists_{P_c \in family} : [H_0(P_a) \bigcup \{P_a\} = H_0(P_c)]]$ then H_0* snowballs. *The notion of a* **USES** *clause telescoping is defined similarly. A partition is* induced *by a telescoping clause c_0 if two processors are in the same partition whenever the sets defined by c_0 overlap.*

Theorem 3.4 *If a* **HEARS** *clause H_0 snowballs, it can be replaced by another* **HEARS** *clause that only specifies input from a single processor.*

Proof: Consider the family of processors described by the **PROCESSORS** declaration that contains the **HEARS** clause. Consider also the induced partition Π. If the cardinality of an equivalence class $E \in \Pi$ is (say) c, then $\forall P_x \in E : \|H_0(P_x)\| < c$. (No processor can **HEAR** itself because it would never be able to complete its calculation if it needed its own result to do so.) Since $\forall x, y : x \neq y \Rightarrow \|H_0(P_x)\| \neq \|H_0(P_y)\|$, and since $\|\{0 \ldots c-1\}\| = c$, the processors in E can be completely ordered by the cardinalities of their **HEAR**d sets. By the basic observation and the snowballing property, each processor can get the information that H_0 requires from the processor that is its predecessor in this ordering. ∎

Definition 3.5 *We call replacing a* **HEARS** *clause, as in the previous theorem,* reducing *the clause.*

The following is proven in Appendix Section B.

Theorem 3.5 *Reducing a snowballing* **HEARS** *clause will produce a parallel structure whose asymptotic speed is the same as the speed of the original structure.*

We can now state this rule in English as follows: "If a **HEARS** clause snowballs then reduce it", and more formally as follows:

rule REDUCE–HEARS (*stmt*) **TRANSFORM**
stmt = '*PNAME* **istype PROCESSORS** (*$PDV*) *$PENUMER*...
if *COND*1 **then**
HEARS $PNAME_{\$HBV}$ *$HENUMER*
(**USES** $UV_{\$UBV}$*$UEN*,...)...'
$\wedge$ (**THEOREM**
$(IS1 = \{HBV : HENUMER[PDV \setminus PDV_1]\}$
$\wedge\, IS1a = \{HBV : HENUMER[PDV \setminus PDV_2]\}$
$\wedge\, IS2 = \{PDV : HENUMER \wedge HBV = PDV\}$
$\wedge\, PROC1 = \{PDV_1\}$
$\wedge\, PROC2 = \{PDV\}$
$\wedge\, PROCh = \{HEXPR\}$
$\wedge\, PROChi = \{HIEXPR\}$
$\wedge\, (IS1 \cap IS1a) \in \{\emptyset\ IS1\ IS1a\}$
$\wedge\, ((\emptyset \subset IS1 \subset IS1a \wedge COND1)$
$\Rightarrow IS1 \cup PROC1 = IS2)$
$\wedge\, (COND2 \leftrightarrow COND1 \wedge IS1 \cup PROCh = IS2)$
$\wedge\, (\sim COND2 \Rightarrow \nexists PDV_3[IS1 \subset \{HBV :$
$HENUMER[PDV \setminus PDV_3]\}])$
$\wedge\, (COND3 \leftrightarrow COND2 \wedge COND1[PDV \setminus HIEXPR])$
$\wedge\, (HIEXPR[PDV \setminus HEXPR] = PDV)))$
$\rightarrow$
stmt = '*PNAME* **istype PROCESSORS** (*$PDV*) *$PENUMER* ...
if *COND*3 **then**
LINKS $PNAME_{\$HEXPR}, PNAME_{\$HIEXPR}$
(**PASSES** $UV_{\$UBV}$*$UEN*,...)
if *COND*2 **then**
HEARS $PNAME_{HEXPR}$ ... '

when this rule is applied to the current state, the bindings will be as follows:

$$
\begin{aligned}
** = \text{`}&\textbf{PROCESSORS } P_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1 \\
&\ldots \\
&\textbf{HEARS } P_{l+k,m-k}, 1 \le k \le m-1 \ldots \text{'} \\
PNAME &= \text{`}P\text{'} \\
PDV &= \text{`}l,m\text{'} \\
PENUMER &= \text{`}1 \le m \le n, 1 \le l \le n-m+1\text{'} \\
HBV &= \text{`}l+k, m-k\text{'} \\
HENUMER &= \text{`}1 \le k \le m-1\text{'} \\
SET1 &= \{(l_1+k, m_1-k) : 1 \le k \le m_1-1\} \\
SET1a &= \{(l_2+k, m_2-k) : 1 \le k \le m_2-1\} \\
PROC1 &= \{(l_1, m_1)\} \\
SET2 &= \{(l+k, m-k) : 1 \le k \le m-1\} \\
PROC2 &= \{(l,m)\} \\
PROCh &= \{(l+1, m-1)\} \\
HEXPR &= \text{`}(l+1, m-1)\text{'} \\
COND1 &= \text{`}2 \le m \le n\text{'} \\
COND2 &= \textbf{true}
\end{aligned}
$$

THEOREM is a function whose argument is a symbolic set-theoretic expression whose atomic terms are set expressions. These expressions are principally created by the **BOUNDBY** function, whose inputs are the bound variables list of the processor name id, an identity parameter, the form that defines the array references that comprise the array definition, and the enumerator (if any) for the array reference.

This rule reduces the **HEARS** clauses from the large **PROCESSORS** declaration of the current state to

$$\textbf{HEARS } P_{l,m-1}$$

$$\textbf{HEARS } P_{l+1,m-1}$$

The resulting **PROCESSORS** declaration is

P **istype PROCESSORS** $(l,m), 1 \leq m \leq n, 1 \leq l \leq n-m+b1$
HAS $A_{l,m}$
if $m = 1$ **then HEARS** Q **(USES** v_l**)**
if $2 \leq m \leq n$ **then**
HEARS $P_{l,m-1}$ **(USES** $A_{l,k}, 1 \leq k \leq m$**)**
HEARS $P_{l+1,m-1}$ **(USES** $A_{l+k,m-k}, 1 \leq k \leq m$**)**

Figure 3.3: Final Form of Main Processors Declaration in P-time Dynamic Programming Derivation

3.1.2.2 Rule A5: Write the Individual Processors' Programs

The general idea of the rule is that the first rule isolated the deepest enumeration in the specification which assigned a value to an array element, and built the beginnings of a parallel structure where each array element within the domain of that enumeration had its own private processor. Since the enumeration in time has been replaced by an enumeration in space, the layers of enumeration that get us to the point which induces the creation of the first parallel structure can be stripped away.

A technical note is that the enumerations can only be completely discarded when there is no calculation at intermediate levels. If there is such calculation, the system will have to add it to the appropriate processors when it strips away the layers of enumeration that include such calculation as well as the deeper enumeration. This does not make the asymptotic behavior of the parallel structure any slower except when the calculations include enumerations. When this is the case, it might be possible to respecify the problem to have separate copies of the array enumerated in the calculation for each cell of the target array. This would require an array whose dimension is the sum of the dimensionalities of the two arrays.

This rule is as follows: "Supply each processor specified by a **PROCESSORS** declaration with a copy of those enumerations from the original program that occurred within the region that included the assignment to array elements that generated that **PROCESSORS** declaration. The references to array elements are replaced by associative lookups from the table of information that the processor has **HEAR**d. The outer enumerations are stripped from the program, and uses of the variables that were bound in these outer enumerations are replaced by constants reflecting the processor's ID."

The derivation of the P-time dynamic programming parallel structure is almost complete. It remains only to reduce the depth of enumeration to the single level implicit in the segment,

$$A_{l,m} = \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$$

Rule A5 does this. The complete parallel structure that results is as follows:

A **istype ARRAY** $(l,m), 1 \le m \le n, 1 \le l \le n-m+1$
P **istype PROCESSORS** $(l,m), 1 \le m \le n, 1 \le l \le n-m+1$
HAS $A_{l,m}$
if $m = 1$ **then HEARS** Q **(USES** v_l**)**
if $2 \le m \le n$ **then**
HEARS $P_{l,m-1}$ **(USES** $A_{l,k}, 1 \le k \le m$**)**
HEARS $P_{l+1,m-1}$ **(USES** $A_{l+k,m-k}, 1 \le k \le m$**)**
v **istype INPUT ARRAY** $(l), 1 \le l \le n$
Q **istype PROCESSORS HAS** $v_l, 1 \le l \le n$
O **istype OUTPUT ARRAY**
R **istype PROCESSORS HAS** O
(in $P_{l,1}, 1 \le l \le n$) :
$A_{l,1} \leftarrow v_l$ $\theta(1)$
(in $P_{l,m}, 2 \le m \le n, 1 \le l \le n-m+1$) :
$A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$ $\theta(n)$
(in $P_{1,n}$) :
$O \leftarrow A_{1,n}$ $\theta(1)$

3.1.2.3 Rules *HEAR-BY-CHAIN*, *SENDS-BY-CHAIN*: Improve Topology of Input/Output

We see that the rules described so far will produce a parallel structure in which every processor is directly connected to the input and output processors when given a specification of array multiplication. Only one I/O processor is created per I/O array, and for many problems, including array multiplication, it is necessary to get some input or output from/to every processor. (P-time dynamic programming is an exception, in which only $\theta(n)$ of the $\theta(n^2)$ processors receive input values and the output is only a single value.)

We therefore conceived another rule to attempt to reduce the excessive connectivity that results from every processor needing access to input or output.

If the following conditions are met:

- the number of processors n_i in a family that receives input from or sends output to a given processor is asymptotically unacceptable, and
- there is a **HEARS** clause H_0 such that the number of processors that do not **HEAR** any processor using H_0 clause (if input) or that are not **HEAR**d by any processor using that clause (if output) is asymptotically less than n_i,

then the I/O **HEARS** clauses can be reduced so that only those processors at a source (or terminus if output) of H_0 are directly connected to the I/O processor.

Pictorially, we convert structures that look like these:

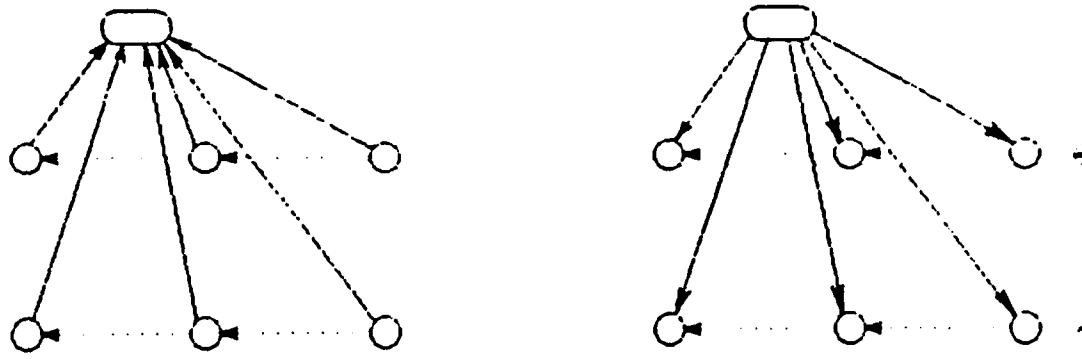


Figure 3.4: Many Processors Use or Build the Same Data

into these:

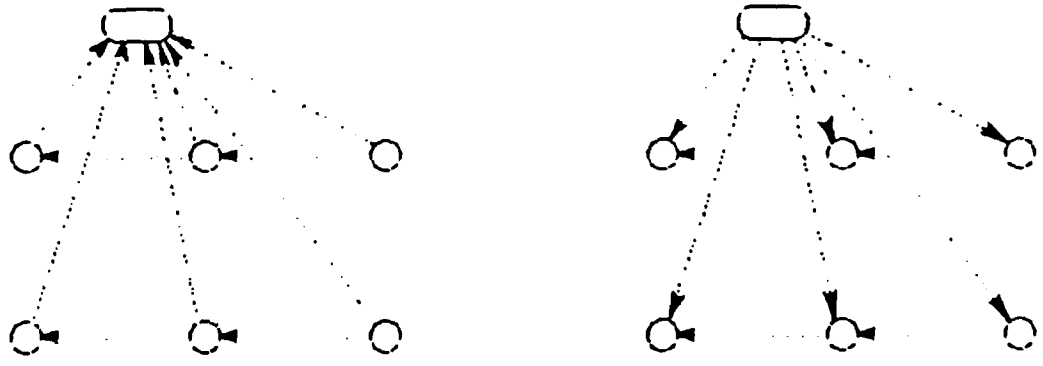


Figure 3.5: Resulting Structure From Sharing I/O Connections

We are not, however, prepared to do this yet. We need to have the chains of processors required by this rule in order to improve the connections to the I/O processors. For this we must introduce another definition and another rule.

3.1.2.4 Rule *MAKE–CHAIN*: Create Interconnections in a Family to Reduce I/O Connectivity

Rules *HEAR–BY–CHAIN*, etc. allow the reduction of connections from/to an I/O processor where a set of interconnections already exists to solve the I/O-free portion of the

problem. In some problems, including array multiplication, no convenient set of interconnections exists and one must be introduced solely to distribute I/O values. Fortunately, the rule that would do this is fairly simple to state and is evidently implementable, given the mechanisms already required for *REDUCE–HEARS*. First we extend Definition 3.7 of induced sets to **USE**d values (the original definition covers only **HEAR**d processors, but it extends in the obvious manner to **USE**d, **SEN**t and **PASS**ed variables as well as **TALK**ed and **PASS**ed to processors.). We then define the notion of telescoping, which heuristically describes a situation in which a set of processors can be split into subsets such that the subsets share interest in a restricted portion of the I/O data.

Definition 3.6 *A clause* `telescopes` *if its induced sets for two processors are either disjoint, or one contains the other. Whenever a clause telescopes, that clause defines a partition, the* `induced partition`*, where two processors are in the same partition element iff the induced sets for the two processors have a non-empty intersection. (Without loss of generality we consider only non-empty sets. Empty sets are not considered because they impose no interconnection requirements.)*

The rule is: where a single **USES** clause telescopes, order the induced partition by the processor indices and interconnect the processors in each partition with a new **HEARS** clause where each processor is connected (only) to its immediate predecessor (if any) in this ordering. Place the **USES** clause within the new **HEARS** clause instead of within the old one.

3.2 A Derivation of a Fast, Parallel Matrix Multiplication Structure

Many parallel structures for the matrix multiplication problem have been proposed in the literature, probably because of the problem's practical importance and its obvious suitability for parallel processing. One of the prettiest parallel structures is described

in [KuL76]. Kung's structure multiplies two $n \times n$ matrices in $\Theta(n)$ time using $\Theta(n^2)$ processors of constant size. (Kung makes the assumption that a solution that involves $\Theta(n)$ processors in communication with the outside world is acceptable. This subsection follows that assumption, which is obviously necessary to multiply $n \times n$ matrices in $\Theta(n)$ time.) There are sequential algorithms with sub-cubic execution time, but there are no obvious mappings of these algorithms into parallel structures.

Some new techniques must be introduced to derive the systolic array of [KuL76]. This will be the subject of the next Section. It is, however, possible to derive a different parallel structure with linear execution time. We added rule *MAKE–CHAIN* with this derivation in mind, but do not feel that *MAKE–CHAIN* is contrived or impractical.

Our parallel structure uses more processors than a systolic array on a restricted class of matrices called "band matrices," in which all but a narrow diagonal band of the input matrices (and therefore of the output matrices) contains zero values. It does, however, use fewer processors on general matrices.

The starting point of this derivation is a specification of matrix multiplication (we are assuming square matrices to simplify the discussion):

A istype **INPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
B istype **INPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
C istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
D istype **OUTPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$

$\forall\, i \in \{1 \ldots n\}$	$\Theta(1)$
$\quad \forall\, j \in \{1 \ldots n\}$	$\Theta(n)$
$\quad\quad C_{i,j} = \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$	$\Theta(n^3)$
$\quad\quad D_{i,j} = C_{i,j}$	$\Theta(n^2)$

The use of arrays C and D seems redundant, but its purpose is technical – our rules would not permit us to assign multiple processors to a single array if that array were an **INPUT** or **OUTPUT** array. Duplicating all of the arrays in this manner, to avoid all

appearances of "prejudicing the case" of which array's parallelism would be important, would change the resulting parallel structure only py replacing each processor by a set of three processors.

MAKE–PSs and *MAKE–IOPSs* add **PROCESSORS** declarations,

A istype **INPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
| PA istype **PROCESSORS HAS** $A_{l,m}, 1 \le l \le n, 1 \le m \le n$
B istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
| PB istype **PROCESSORS HAS** $B_{l,m}, 1 \le l \le n, 1 \le m \le n$
C istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
| PC istype **PROCESSORS** $(l,m), 1 \le l \le n, 1 \le m \le n$ **HAS** $C_{l,m}$
D istype **OUTPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
| PD istype **PROCESSORS HAS** $D_{l,m}, 1 \le l \le n, 1 \le m \le n$

$\forall i \in \{1 \ldots n\}$ $\Theta(1)$
  $\forall j \in \{1 \ldots n\}$ $\Theta(n)$
    $C_{i,j} = \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$ $\Theta(n^3)$
    $D_{i,j} = C_{i,j}$ $\Theta(n^2)$

MAKE–USES–HEARS completes the rough form of these declarations.

A istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
PA istype **PROCESSORS HAS** $A_{l,m}, 1 \le l \le n, 1 \le m \le n$
B istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
PB istype **PROCESSORS HAS** $B_{l,m}, 1 \le l \le n, 1 \le m \le n$
C istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
PC istype **PROCESSORS** $(l,m), 1 \le l \le n, 1 \le m \le n$ **HAS** $C_{l,m}$
| **HEARS** PA(**USES** $A_{l,k}, 1 \le k \le n$)
| **HEARS** PB(**USES** $B_{k,m}, 1 \le k \le n$)
D istype **OUTPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
PD istype **PROCESSORS HAS** $D_{l,m}, 1 \le l \le n, 1 \le m \le n$
| **HEARS** $\mathrm{PC}_{l,m}, 1 \le l \le n, 1 \le m \le n$
(**USES** $C_{l,m}, 1 \le l \le n, 1 \le m \le n$)

$\forall i \in \{1 \ldots n\}$ $\Theta(1)$
  $\forall j \in \{1 \ldots n\}$ $\Theta(n)$
    $C_{i,j} = \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$ $\Theta(n^3)$
    $D_{i,j} = C_{i,j}$ $\Theta(n^2)$

REDUCE–HEARS is unable to improve this parallel structure, because there are no interconnections among the PCs to improve. Rule *HEAR–BY–CHAIN* is also helpless, although the topology of the interconnection graph is too rich ($\Theta(n^2)$ rather than the goal of $\Theta(n)$). Rule *MAKE–CHAIN* comes to the rescue. Adding the **HEARS** clauses allowed by *MAKE–CHAIN* and by the **USES** clauses of PC produces:

A **istype ARRAY** $(l,m), 1 \le l \le n, 1\ lem \le n$ **INPUT**
PA **istype PROCESSORS HAS** $A_{l,m}, 1 \le l \le n, 1 \le m \le n$
B **istype ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
PB **istype PROCESSORS HAS** $B_{l,m}, 1 \le l \le n, 1 \le m \le n$
C **istype ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
PC **istype PROCESSORS** $(l,m), 1 \le l \le n, 1 \le m \le n$ **HAS** $C_{l,m}$
HEARS PA(**USES** $A_{l,k}, 1 \le k \le n$)
HEARS PB(**USES** $B_{k,m}, 1 \le k \le n$)
if $m > 1$ **then HEARS** $\mathrm{PC}_{l,m-1}$
if $l > 1$ **then HEARS** $\mathrm{PC}_{l-1,m}$
D **istype OUTPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
PD **istype PROCESSORS HAS** $D_{l,m}, 1 \le l \le n, 1 \le m \le n$
HEARS $\mathrm{PC}_{l,m}, 1 \le l \le n, 1 \le m \le n$
(**USES** $C_{l,m}, 1 \le l \le n, 1 \le m \le n$)

$\forall\, i \in \{1 \ldots n\}$ $\Theta(1)$
$\forall\, j \in \{1 \ldots n\}$ $\Theta(n)$
$C_{i,j} = \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$ $\Theta(n^3)$
$D_{i,j} = C_{i,j}$ $\Theta(n^2)$

Then rule *HEAR–BY–CHAIN* is applied twice, and rule *SEND–BY–CHAIN* once, finishing the derivation.

A **istype ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
PA **istype PROCESSORS HAS** $A_{l,m}, 1 \le l \le n, 1 \le m \le n$
B **istype ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
PB **istype PROCESSORS HAS** $B_{l,m}, 1 \le l \le n, 1 \le m \le n$
C **istype ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
PC **istype PROCESSORS** $(l,m), 1 \le l \le n, 1 \le m \le n$ **HAS** $C_{l,m}$
if $m = 1$ **then HEARS** PA(**USES** $A_{l,k}, 1 \le k \le n$)
if $l = 1$ **then HEARS** PB(**USES** $B_{k,m}, 1 \le k \le n$)
if $m > 1$ **then HEARS** $\mathrm{PC}_{l,m-1}$(**USES** $A_{l,k}, 1 \le k \le n$)
if $l > 1$ **then HEARS** $\mathrm{PC}_{l-1,m}$(**USES** $B_{k,m}, 1 \le k \le n$)

D istype **OUTPUT ARRAY** $(l,m), 1 \leq l \leq n, 1 \leq m \leq n$
PD istype **PROCESSORS HAS** $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
USES $C_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
HEARS $\mathrm{PC}_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$

| (in $\mathrm{PC}_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$) :
| $C_{l,m} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{l,k} B_{k,m}$ $\Theta(n)$
| (in PD) :
| $D_{l,m} \leftarrow C_{l,m}$ $\Theta(1)$

3.3 Virtualization and Aggregation

3.3.1 An Informal Description

In virtualization we select a variable (which might be a hidden variable such as the accumulation variable in a reduction operation) that receives assignments in a loop, explicate it if necessary, and provide it with enough indices to meet the single assignment condition. If this is done as often as possible to a specification, the value of each variable will depend on a constant amount of computation, independent of the problem size. Since the size of the virtualized structure will not be independent of the problem size, and since communication is such that the running time of the problem will be polynomial in that size, a fully virtualized structure will make light use of each processor.

We can make use of this fact either by pipelining or aggregation.

Consider the enumeration:

$$c_{ij} = \sum_{k \in \{1,\ldots,n-1\}} a_{ik} b_{kj}$$

There is an enumeration, but only over values, not destinations. For this reason, use of separate processors will not be generated for the steps of the enumeration. 1 Now one can make a few changes to the specification in order to generate separate processors for the steps of the enumeration. (We will see the need for separate processors below.)

Generate the following *virtualization*, creating the array C

c **istype ARRAY** $(\{1,\ldots,n\},\{1,\ldots,n\},\{0,\ldots,n\})$

. . .

$c_{l,m,0} = 0$
$\forall\ k \in (1,\ldots,n)$ **do**
$c_{l,m,k} = c_{l,m,k-1} + a_{ik}b_{kj}$

This structure represents several changes:

- First, it introduces a new dimension to the main array for each level of enumeration performed to find a value for the old elements of the array.
- Second, the enumeration $k \in \{1 \ldots n\}$ into the enumeration $k \in (1 \ldots n)$ is changed. This is perfectly legitimate—the set enumeration does not *forbid* enumeration in a specified order. When we consider automating this process, however, we should remember that there are $n!$ ordered enumerations corresponding to a specific unordered one of length n. The best orderings to try will probably include the arrival orderings inferrable from **HEARS** and **HAS** clauses, and the "natural" orderings, i.e., numerical order and inverse numerical order (where numbers are involved).

  Of course, this only applies when the inner enumeration(s) enumerate over a set. When the enumerand is already a sequence, this step is unnecessary.
- An identity for the enumeration's operation is selected. This can be artificial, a special null value that is checked for.
- Fourth, an ordering for the enumeration is selected.
- Fifth, explicit code to create running totals is generated.

In aggregation, instead of multiplying the number of processors we group processors into one. We require that each processor have insufficient work to occupy the time that the parallel structure requires for solution. This can arise because the processors need to wait for partial results from other processors.

When this is the case, the processors can be collected into groups that don't share any processing times. Each group is then replaced by a single processor that is responsible for all of the values computed by any member of the group.

Aggregation is needed to get efficient parallel structures after virtualization, because virtualization produces specifications that would be transformed into structures that have a constant amount of work per processor. This is much less than the amount of time available to the system, because after virtualization there must be chains of processors whose length is linear in some measure of the size of the problem. Heuristically, virtualization makes too many processors and aggregation is necessary to undo this.

The power of the techniques arises from their ability to together arrange for different parts of the work of computing a single element of the answer to be performed in different processors.

3.3.2 Definitions of Virtualization and Aggregation

Definition 3.7 *A* `virtualization` *of a parallel structure is a new parallel structure that results from*

- adding a dimension to an array, say A, producing A as follows: if A_I is a defined element of A, and the computation of A_I is performed by enumerating n elements of some set or vector S and performing a binary operation on a running total and each element of S as it is enumerated, then $A_{I|m}$ for $0 \leq m \leq n$ will be a defined element of the new array, A;

- making the enumeration of S an ordered one; and
- replacing the original enumeration/calculation with a calculation that explicitly folds the j^{th} value of the ordered enumeration as performed for A_I by operating on $A_{I|j-1}$ and that j^{th} element.

The process of creating a virtualization is also called virtualization.

Definition 3.8 *An* **aggregation** *of a parallel structure is a new parallel structure that results from partitioning the old set of processors of a family into equivalence classes, and creating a processor for each equivalence class. A processor in the aggregation* **HEAR***s another such processor if any processor in the first equivalence class* **HEAR***d any processor in the second.*

The process of creating an aggregation is also called aggregation.

There are, of course, an intractable number of possible aggregations according to this definition. Only simple aggregations are worthy of consideration, because allowing complex ones would lead to a combinatorial explosion; also, the complex ones would tend either to leave too many interprocessor connections or to have too much work being done in some of the processors.

Suppose that the unaggregated family of processors is defined as

$$\mathrm{P}_{x_1,x_2,\ldots x_m}, \langle enumers \rangle$$

We will use P_X to represent this below. We use a notation for aggregations that describes sets of processors that are identified by the aggregation. As an example, suppose we have an $m \times n$ array of processors with a family name P, and we want to create a processor

family with n processors, one from each column. The one from column i will be Q_i. The notation in

$$\mathbf{Q\ istype\ AGGREGATION}\ (i), 1 \leq i \leq n-1; \{P_{ji} : 1 \leq j \leq m-1\} \ldots$$

(where the usual **HAS**, **HEARS**, **TALKS** follows).

The aggregations we will consider for a family of processors organized as an d-dimensional array whose bounds are the vectors L and U can be categorized as follows: Consider the d element vectors each of whose elements is either -1, 0 or 1 and some of whose elements are non-zero. We will consider an aggregation that produces a new processor for each set of processors $\{P_{X+jI} : L \preceq X + jI \preceq U\}$ where $L \preceq X$ has the usual meaning of $\forall 1 \leq j \leq d[l_j \leq x_j]$.

3.3.3 Systolic Structure Synthesis

We now study distributional problems preferably implemented by a systolic array. For a specific example, suppose we are evaluating

$\forall i \in (1, \ldots, m)$; must be enumerated in order
  $\forall j \in \{1, \ldots, n\}$; may be enumerated in any order
    $B_j = B_j + A_i$

and suppose that l and n are of the same order of magnitude, or that all of the B-values are in a single place and we choose not to distribute them. A systolic structure is preferable to a tree.

Consider the structure below:

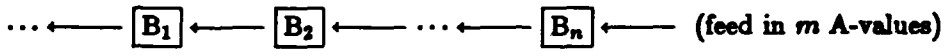


Figure 3.6: Simple Parallel Structure for Broadcasting

in which each of the m A-values is added to each of the n B-values. Note that no source of B-values is given; each processor must be connected to B's I/O processor.

We explicate the m partial sums, using virtualization. This creates a separate processor for each *step* in the summation process for each of the B-values.

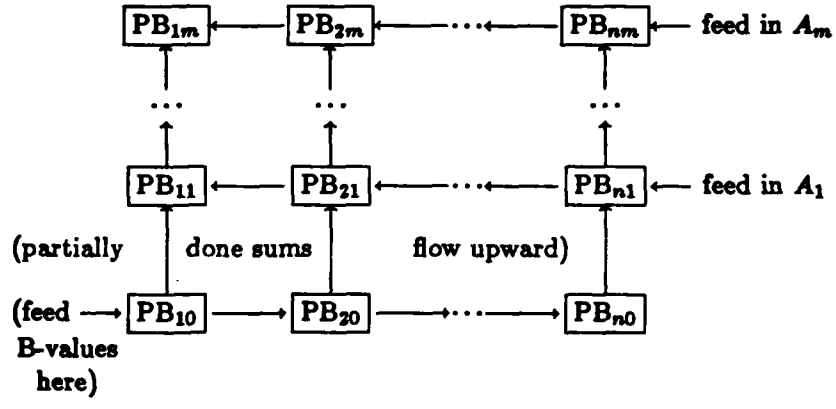


Figure 3.7: Virtualized Broadcast Structure

Then we modify that slightly to feed in A-values in only one place.

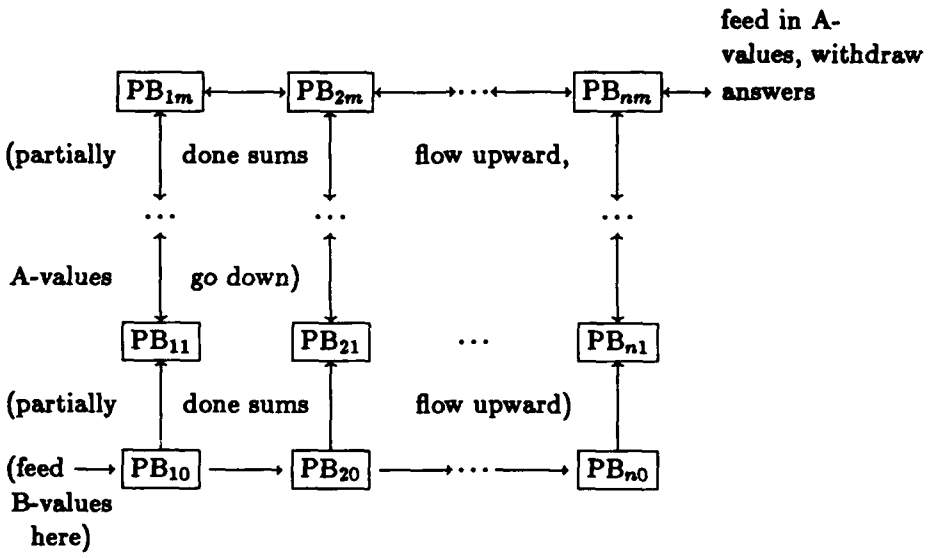


Figure 3.8: Virtualized Broadcast Structure with Chains for I/O

We then identify P_{ij} with $P_{i+k,j-k}$ for all appropriate k:

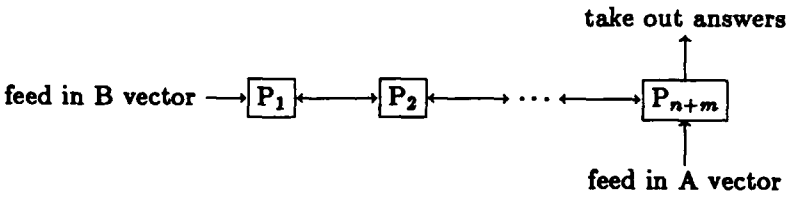


Figure 3.9: Aggregation of Virtualized Broadcast Structure

This parallel structure is better than a structure synthesized directly from the specification because it does not impose strenuous requirements on the I/O capabilities of the system. The specification does not say how B-values get to the various B_i. If this were exposed, we would see that the assumption is made either that the broadcast problem was embedded in a larger problem that allows the data to already be there, or that all n B-processors **HEAR** the I/O processor. The systolic array shown above allows the I/O processors to be connected to only a single processor.

3.3.4 Use of Virtualization and Aggregation for Matrix Multiplication

Consider the case of synthesizing parallel structures for matrix multiplication that work in linear time. Application of the techniques previous to virtualization and aggregation produces the following parallel structure:

A istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
PA istype **PROCESSORS HAS** $A_{l,m}, 1 \le l \le n, 1 \le m \le n$
B istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$ **INPUT**
PB istype **PROCESSORS HAS** $B_{l,m}, 1 \le l \le n, 1 \le m \le n$
C istype **ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
PC istype **PROCESSORS** $(l,m), 1 \le l \le n, 1 \le m \le n$ **HAS** $C_{l,m}$
if $m = 1$ **then HEARS** PA(**USES** $A_{l,k}, 1 \le k \le n$)
if $l = 1$ **then HEARS** PB(**USES** $B_{k,m}, 1 \le k \le n$)
if $m > 1$ **then HEARS** $\mathrm{PC}_{l,m-1}$(**USES** $A_{l,k}, 1 \le k \le n$)
if $l > 1$ **then HEARS** $\mathrm{PC}_{l-1,m}$(**USES** $B_{k,m}, 1 \le k \le n$)
D istype **OUTPUT ARRAY** $(l,m), 1 \le l \le n, 1 \le m \le n$
PD istype **PROCESSORS HAS** $D_{l,m}, 1 \le l \le n, 1 \le m \le n$
USES $C_{l,m}, 1 \le l \le n, 1 \le m \le n$
HEARS $\mathrm{PC}_{l,m}, 1 \le l \le n, 1 \le m \le n$

$$C_{l,m} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{l,k} B_{k,m} \qquad \Theta(n)$$

$$D_{l,m} \leftarrow C_{l,m} \qquad \Theta(1)$$

The asymptotic behavior of this parallel structure seems to be the same as that for Kung's parallel structure [KuL76]. However, there can be an advantage of Kung's parallel structure over the simpler one. When multiplying "band matrices", where $j - i < k_{0,0} \vee j - i > k_{1,0} \Rightarrow A_{i,j} = 0$ and $j - i < k_{0,1} \vee j - i > k_{1,1} \Rightarrow A_{i,j} = 0$, it is possible to use fewer processing elements. If $k_{1,0} - k_{0,0} + 1 = w_0$ and $k_{1,1} - k_{1,0} + 1 = w_1$, then it can be shown that only $(w_0 + w_1)n$ of the n^2 processors of our parallel structure can have non-zero answers, and only that many processors have to be provided. With Kung's parallel structure, however, only $w_0 w_1$ processors have to be provided. The multiplication takes $\Theta(n)$ time. (It is possible to use the $\Theta((w_0+w_1)n)$ processors to multiply the band matrices in $\Theta(w_0 + w_1)$ time, but this parallel structure cannot be synthesized automatically using

these techniques, and in any event the time/processors tradeoff offered by Kung's parallel structure may be desirable.)

The virtualization process, alone, is not enough to synthesize Kung's systolic arrays. Notice that the number of processors in the parallel structure that results from the obvious virtualization is $\Theta(n^3)$. Partial sums of product array elements reside in different processors at different times. This feature makes some technique like virtualization necessary to separate the computation of partial products, but processors have to be grouped to prevent this processor count blowup. Another more difficult technique, *aggregation*, will reduce the processor count to the target level.

Heuristically, aggregation is the grouping together of processors, each of which does a small amount of work, into groups of processors, each represented by a single processor. Each processor does all of the work that any processor in the original group did, but this can still be done quickly because each of the processors in the original group had a small amount of work to do, and no two processors had to do their work at overlapping times.

The reason why Kung's parallel structure can multiply matrices in linear time using constant space per processor is that he has performed a virtualization on the summation of result array elements. He avoids the need for n^3 processors by a process called *processor aggregation*. Each processor is responsible for computing $\Theta(n)$ elements of the virtual array.

Reasoning similar to that performed in the change-of-basis generator and theorem prover will serve us well here. The target interconnection structure is

P **istype PROCESSORS** $(l,m), -n \le m \le n, -n \le l \le n-m+1$
HAS $C_{i,j,k}, 1 \le i \le n, 1 \le j \le n, 1 \le k \le n, i-j=l-m, k=\min(n-l+1, n+m+1, n)$
if $m < n$ **then**
HEARS $P_{l,m+1}$ **(USES** $A_{i,j}, 1 \le i \le n, 1 \le j \le n, i-j=l$**)**
if $-n < m < n$ **then**
LINKS $P_{l,m+1}, P_{l,m-1}$ **(PASSES** $A_{i,j}, 1 \le i \le n, 1 \le j \le n, i-j=l$**)**
if $l > -n$ **then**
HEARS $P_{l-1,m}$ **(USES** $B_{i,j}, 1 \le i \le n, 1 \le j \le n, i-j=m$**)**
if $n > l > -n$ **then**
LINKS $P_{l-1,m}, P_{l+1,m}$ **(PASSES** $B_{i,j}, 1 \le i \le n, 1 \le j \le n, i-j=m$**)**
if $l < n \wedge m > -n$ **then**
HEARS $P_{l+1,m-1}$
(USES $C_{i,j,k-1}, 1 \le i \le n, 1 \le j \le n, 1 \le k \le n, i-j=l-m, k=$
$\min(n-l+1, n+m+1, n)$**)**
if $l > -n \wedge m < n$ **then**
TALKS $P_{l-1,m+1}$
(SENDS $C_{i,j,k}, 1 \le i \le n, 1 \le j \le n, 1 \le k \le n, i-j=l-m, k=$
$\min(n-l+1, n+m+1, n)$**)**

which is Kung's structure. This requires two changes of basis of input matrices ($i-j$ of both A and B, rather than either i or j), and a change of basis for the C array, as well as replacement of the summation of each C-array element over a set of integers with a summation over a sequence of integers.

The figures below illustrate the virtualization and aggregation processes as they apply to an $n = 3$ instance of a matrix multiplication problem.

Figure 3.10 shows the basic topology of the matrix multiplication parallel structure;

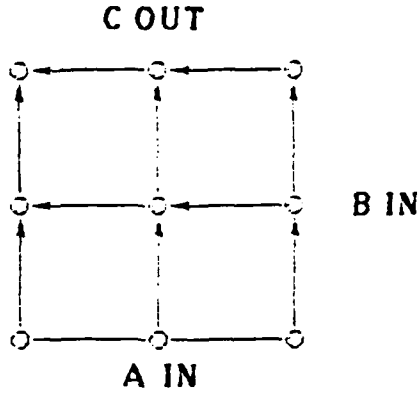


Figure 3.10: Unvirtualized Structure

Figure 3.11 shows the topology after the virtualization has been performed over the summation;

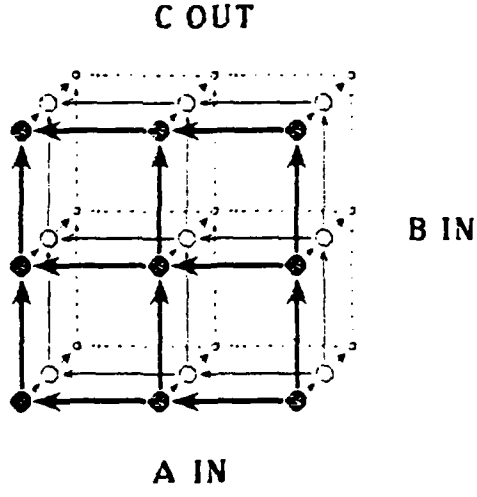


Figure 3.11: Virtualization

Figure 3.12 shows the sets of processors that are to be collapsed into a single processor;

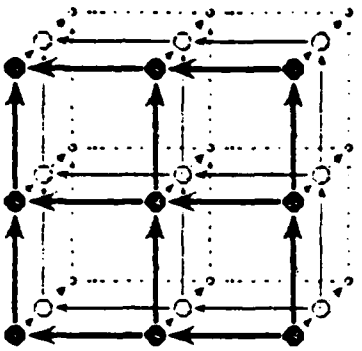


Figure 3.12: Aggregation to be performed

and Figure 3.13 shows the configuration that results, in a form that would be more recognizable to the student of [KuL76].

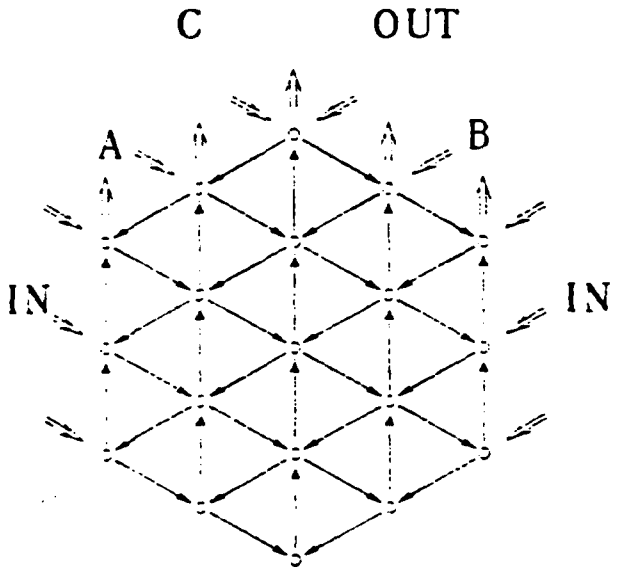


Figure 3.13: Aggregated (Systolic) Structure

3.3.5 What Virtualization Can and Cannot Accomplish

An important measure of the cost of a parallel structure is the product of the number of processors, the size of each one, and the amount of time the parallel structure takes to do a calculation. We will call this the PST measure.

PST = $\Theta((w_0 + w_1)n^2)$ for the simpler parallel structure for matrix multiplication, when applied to band matrices of widths w_0 and w_1. Virtualization and aggregation can improve this to $\Theta(w_0 w_1 n)$ by reducing the number of processors while allowing the size of the processors and the running time of the algorithm to remain the same.

It is possible to achieve PST = $\Theta((w_0 + w_1)^2 n^2)$ by other means. This is equivalent whenever $w_1 = \Theta(w_0)$. Divide the $n \times n$ array of potential processors into $(w_0 + w_1) \times (w_0 + w_1)$ blocks and introduce input and output connections at the appropriate edges of each such block. This is impossible to derive by techniques shown so far, or reasonable extensions to them. It has the further disadvantage that the number of connections to input and output processors is $\Theta(n)$, while the same number is $\Theta(w_0 w_1)$ for the systolic array parallel structure that results from virtualization and aggregation. A complexity measure that took into account the connections to the I/O processors would favor the systolic array structure even over the improved simple matrix multiplication scheme.

It should be noted that the parallel structure resulting from partitioning the potential processors has the same PST as systolic arrays, but P and T are different. Different measures, such as PST^2 may make different parallel structures more desirable.

3.4 User-Assisted Aggregation

We have an arbitrary set of aggregations we will consider in order to keep the size of the algorithm search tree reasonable, but there may be cases where it would be advisable to consider other aggregations. For this reason, when the system considers an aggregation it codes this information without hidden information, allowing the user to insert **AGGREGATE** declarations of his own. The reason we chose to allow this rather than to make an attempt to have the aggregation finding machinery be complete are:

- The bounds of arrays are often arbitrary.

- There are many aggregations available; it is not clear which are useful.
- It is not uncommon for two logical data to share parts of an array.
- It is possible for one array to match the combination of two others.

For these reasons, TRANSCONS understands the **AGGREGATION** type, of the form:

Name **istype AGGREGATE** (*bound*) *iters* ≡ $Pset_{bound}$
HAS $Aset_{bound}$
(**HEARS** $Pname2_{P(bound)}$ $iters2_{bound}$
(**USES** $Aset2_{bound}$ ...))
HAS ...;

This statement is a parameterized statement. The *iters* is a predicate defining permissible bindings of the variables in the list *bound*. It means that those processors in the set $Pset_{bound}$ (which is a set-valued expression) are aggregated (i.e., identified to form a single processor named $Name_{bound}$) for each binding of the variables in *bound* that is permitted by the predicates in *iters*. It is explicitly permitted that the set-valued expression can include enumerated elements and explicit setformers.

The **HAS**, **HEARS** and **USES** elements are analogous to those of a **PROCESSORS** declaration (see above), although in an **AGGREGATION** type there can be more than one **HAS** clause. **HEARS** clauses are associated with specific **HAS** clauses.

When the user provides such assistance, searching a potentially enormous set of possible interfamily aggregations is avoided. TRANSCONS's abilities are used to check the validity of the user-proposed aggregation.

The following consistency checking is performed on **AGGREGATION** declarations:

(i) formally specified conditions:

- *Pset* is disjoint for all distinct settings of bound and for all settings of the respective bounds for two **AGGREGATION** declarations.

- *Name*(⟨specific bound⟩) **HAS** ⟨array element⟩ iff $\exists P(F) \in Pset$(⟨specific bound⟩) that **HAS** ⟨array element⟩.

if *Pname2* = *Name*, **then**
$\forall bound$ **s.t.** *iters*, $bound2 = F(bound)$, $bound \neq bound2$:
$\exists P_{b3} \in PSET_{bound}$, $P_{b4} \in \text{Pset}_{\text{bound2}}$:
P_{b3} **HEARS** P_{b4}(**USES** A)

(meaning that the **HEARS** clauses of the **AGGREGATION** declaration are those induced by the underlying processors);

(ii) informally specified conditions:

- The order of total amount of computation done by processors underlying a given node does not exceed the length of the longest chain.

- It is not true that *A* **HEARS** *B* and *B* **HEARS** *A* for the same **USES** datum. (But violation of this condition is likely to imply violation of others)

It is important to note that a user's aggregation declarations are indistinguishable (to the system) from automatically inserted ones.

Chapter 4

Trees, Closures and Divide & Conquer or, LAMBDA: The Ultimate Transceiver*

4.1 Motivation

Suppose information must flow from processor B to processor A, but there is a conceptual advantage to viewing the problem as if information were flowing the other way. We have two motivating situations in which this is the case. One is the handshake problem, in which an intermediate processor in a pipelined chain of processors must be able to declare its readiness to handle another datum after it has processed a first. The second is problems requiring tree-structured collections of interconnected processors. We would like to use divide & conquer (D&C) to synthesize these trees, but that technique is difficult to apply if data conceptually flow both up and down the tree. It becomes easier if the flow is conceptually one way. We claim that D&C is a powerful synthesis technique that can produce a large class of tree structured architectures if problems can be rephrased in terms of one-way data flow.

We want to bring about a structure in which information flowing from a processor A to another processor B tells B what to do with other information computed in B but needed

* With apologies to Guy Steele [Ste77]

by A. We then want to restrict our synthesis task to reasoning about, computing and sending the datum from A to B. We need a new type of datum, the *closure*. Processor A sends processor B a closure, and B can later use it to cause data to be sent back to A and to be used properly. We explore a series of requirements that must be met for D&C to be used to synthesize tree structures. We show that if these requirements are met we get efficient tree parallel structures. We then describe closures, including some technical issues that make plausible that they are implementable in a reasonable computation model. This allows us to show that the D&C requirements are not restrictive, provided we allow closures, by exhibiting a constructive proof that there is a structure that meets our requirements equivalent to any member of a broad class of structures that do not. We end this Chapter by exhibiting the syntax we use to describe trees and some formal axioms about them.

4.2 The Divide & Conquer Paradigm and Tree Synthesis

D&C is a widely used technique for the synthesis of single-processor programs, and one feels that it should be a good technique for the synthesis of tree-shaped parallel structures. Trouble often arises, however, when we try to use D&C for this purpose.

Consider what the D&C technique actually is. "To solve a 'large' problem instance, break it into pieces, solve the problem for each of the pieces, and combine the solutions". This is a technique for generating $O(n)$ and $O(n \log n)$ time, single processor solutions to a wide variety of problems. See, for example, [**Smi83**], [**Knu69**] and [**AHU74**].

Intuition would lead us to believe that D&C is useful for synthesizing tree-structured parallel structures, because the structure of a solution closely matches the structure of the desired set of processors. Three classes of problems arise, however:

- **rootlock**: When we try to combine solutions for subproblems, the amount of information traveling either from one subproblem to another or from the subproblems to the combination operator, or the amount of work necessary to combine, may be asymptotically large in the problem size. A naïvely synthesized parallel structure would have to perform all of this work in one processor, namely a "root" processor that has responsibility for combining partial solutions into a solution to the whole problem.

- **sequentiality**: In a variant of D&C, one solves one of the subproblems *first*, and uses some function of the solution as a parameter to the process that takes place elsewhere. It is clear that in this case no problem element can enter the computation until all previous elements have been used. There is no concurrency.

- **bidirectionality**: Information might have to flow both up and down the tree to obtain a solution. This situation can make formal description of a combination operator for D&C hard. It might appear that this condition is intrinsic to D&C, but that is not the case. The data could already be distributed among an array of processors (or available to be so distributed) and the division step can manipulate indices only.

It is possible to have bidirectionality without sequentiality, but not *vice versa*. Rootlock is independent of the other two situations.

These three properties of D&C solutions to specifications are impediments to easy synthesis of tree-structured parallel structures for these specifications.

A specification, three of whose natural D&C solutions have one of these features each, is Prefix Summation. In this specification, we have a one dimensional vector A of length n, and we want to create a vector A such that $\forall 1 \leq i \leq n[a_i = \sum_{1 \leq j \leq i} a_j]$. In what follows

we use the words "left" and "right" as if the array were arranged in a row with a_1 leftmost and a_n rightmost.

One solution is "to perform prefix summation on a non-trivial vector, divide it into two halves, perform prefix summation on each half, and add the rightmost element of the left result to each element of the right result". This solution has bidirectionality.

A second solution is to first define "augmented prefix summation with augend z" as $\forall 1 \leq i \leq n[a_i = z + \sum_{1 \leq j \leq i} a_j]$. We then say that to perform augmented prefix summation with augend z on a non-trivial vector $a_{l:u}$, divide it into two halves $a_{l:u}$ and $a_{u+1:u}$, perform augmented prefix summation with z on the left half, and perform augmented prefix summation with $z + a_u$ on the right half. This is intrinsically sequential.

A third solution is similar to the first, except that the result vector is carried up the tree as the value of the D&C step rather than having, as the goal, to develop the new values at the leaves. This has *rootlock*, i.e., it is an $O(n)$ solution when implemented on a tree-structured multiprocessor system in a natural manner as it requires funnelling the entire result vector through the root.

In the remainder of this Section we formalize the problems described above. We also show that if a D&C formulation has none of these problems then there exist fast parallel structures that are functionally equivalent. We then exhibit the syntactic structures we use to describe trees. We then describe the closure concept and we argue that use of closures makes the imposition of our requirements much less restrictive than otherwise by showing that a broad class of structures not meeting them can be syntactically transformed into ones that do. Finally we show that use of these closures causes no loss of computation speed (within a constant factor).

In the following, we will describe programs generically using *schemata*, in the usual sense. A scheme is a program fragment in which some operators are merely given names

rather than being described. If all of the names are given an interpretation a scheme becomes a program or a scheme instance or an instance of the scheme. Below we define sequentiality and bidirectionality and a notion, *P-combination*, or a combination operator that makes the scheme suitable for an implementation in which solutions to the subproblems are developed in parallel in separate processors, delivered to a third processor, and combined by that processor.

In the schema that follow, upper case letters with subscripts and superscripts are parametrized functions of appropriate arity. The subscripts and superscripts are integers representing a range of elements from the problem instance, and the actual problem represented by a function in this notation depends on the values of the range parameters and on the values from the problem instance in that range.

Definition 4.1 *A specification is* `P-combinational` *if it is an instance of this scheme:*

$$V_l^u = \begin{cases} W_l, & \text{if } l = u \\ G(V_l^u, V_{u+1}^u), & \text{otherwise} \end{cases}$$

A specification is `sequential` *if it is not P-combinational and it is an instance of one of these schemata:*

$$V_l^u = \begin{cases} W_l, & \text{if } l = u \\ G(V_l^u, H_{u+1}^u(V_l^u)), & \text{otherwise} \end{cases}$$

or

$$V_l^u = \begin{cases} W_l, & \text{if } l = u \\ G(H_l^u(V_{u+1}^u), V_{u+1}^u), & \text{otherwise} \end{cases}$$

A specification is `bidirectional` *if it is not P-combinational or sequential and it is an instance of one of these schemata:*

$$V_l^u = \begin{cases} W_l, & \text{if } l = u \\ G(V_l^u, H_{u+1}^u(V_l^u, V_{u+1}^u)), & \text{otherwise} \end{cases}$$

or

$$V_l^u = \begin{cases} W_l, & \text{if } l = u \\ G(H_l^u(V_{u+1}^u, V_l^u), V_{u+1}^u), & \text{otherwise} \end{cases}$$

The important point of these definitions is that the result of the recursion for one of the parts of the problem after the division step is used as an argument to the function that computes the other part of the result (H_i^j). The treatment of half of the problem depends on the solution of the other half.

There are some lemmas to prove before the main results of this Chapter. In what follows, we will use $T(\langle \text{a value} \rangle)$ or $T(\langle \text{a function} \rangle)$ to denote the time required for an optimal parallel structure to compute the value or the function. $T(\langle \text{a processor} \rangle)$ is the time for that processor to generate its result (where the meaning is obvious). In all cases where we use this notation, it will be obvious how the evaluation is to take place. We also assume that there exists a monotonically increasing function F such that $T(G_l^u) \leq F(u - l + 1)$.

Lemma 4.1 *If we have a D&C scheme instance of the following:*

$$V_l^u = \begin{cases} W_l, & \text{if } l = u \\ G(V_l^u, V_{u+1}^u), & \text{otherwise} \end{cases}$$

then $T(V_l^u) \leq \max(T(V_l^{\lfloor (l+u)/2 \rfloor}), T(V_{\lceil (l+u+1)/2 \rceil}^u)) + O(T(G))$.

Proof: By the definition of $T(\ldots)$, there exist two parallel structures, one that computes $V_l^{\lfloor (l+u)/2 \rfloor}$ in $T(V_l^{\lfloor (l+u)/2 \rfloor})$ and one that computes $V^u_{\lceil (l+u+1)/2 \rceil}$ in $T(V^u_{\lceil (l+u+1)/2 \rceil})$.

Let the processor that develops $V_l^{\lfloor (l+u)/2 \rfloor}$ be called P_l and that which develops $V^u_{\lceil (l+u+1)/2 \rceil}$ be called P_r. Connect a new processor to each of P_l and P_r, and call it P.

P_l and P_r develop their results in $T(V_l^{\lfloor (l+u)/2 \rfloor})$ an $T(V^u_{\lceil (l+u+1)/2 \rceil})$, respectively. The amount of time required to communicate both of these results is

$$
\begin{aligned}
& O\left(\left|V_l^{\lfloor (l+u)/2 \rfloor}\right| + \left|V^u_{\lceil (l+u+1)/2 \rceil}\right|\right) \\
\leq & O(F(\lceil (u-l+1)/2 \rceil)) \qquad \text{(by monotonicity of } F.)
\end{aligned}
$$

This requires the observation that the time required to develop a result is at least proportional to the result's length.

Once the results are both delivered to P, it computes its result in $T(G)$, making the total time for the computation

$$
\begin{aligned}
& \max\left(T(V_l^{\lfloor (l+u)/2 \rfloor}), T(V^u_{\lceil (l+u+1)/2 \rceil})\right) && \text{(develop partials)} \\
& + O(F(\lceil (u-l+1)/2 \rceil)) && \text{(by above observation)} \\
& + O(F(u-l+1)) && \text{(by hypothesis)}
\end{aligned}
$$

and the two last bounds coalesce by the fact that F is monotonically increasing. ∎

We can now prove that the computation of the closure in the root node is fast:

Theorem 4.2 *Suppose a problem fits a D&C scheme with P-combination. That is, that the computation of the result in question for the substring of the problem ranging from l to u is*

$$V_l^u = \begin{cases} W_l, & \text{if } l = u \\ G(V_l^u, V_{u+1}^u), & \text{otherwise} \end{cases}$$

and $T(G)$ (the time to compute G) is $\leq O(F(u-l+1))$, where F is a nondecreasing function. Then $T(V_1^n) = O(F(n)\log n)$.

Proof: Note that the form of the definition of V_l^u precludes sequentiality and bidirectionality. We are using value semantics for the call to G. $T(V_i^i) = T(V)$, so $T(V)$ is bounded. Say $T(G) \leq c_0 F(u-l+1)$. We prove by induction that $T(V_l^u) \leq c_0 F(u-l+1)\lg(u-l+1) + T(V)$, where c_0 is the constant of $T(V_1^n) = O(F(n)\log n)$.

The base case is immediate.

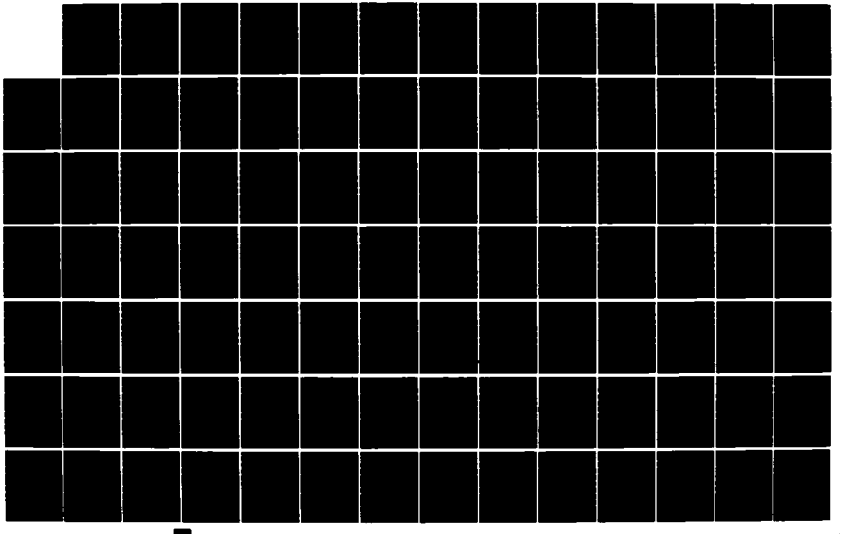
If $l \neq u$ then

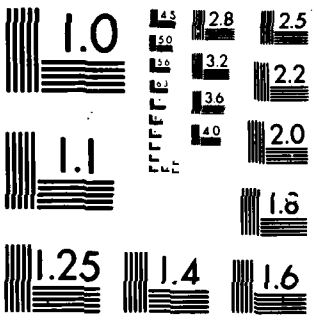
$$\begin{aligned} T(V_l^u) &= \max\left(T(V_l^{\lfloor (l+u)/2 \rfloor}), T(V_{\lceil (l+u+1)/2 \rceil}^u)\right) + T(G) && \text{(definition, nonsequentiality)} \\ &\leq c_0 F((u-l+1)/2)\lg((u-l+1)/2) + T(V) + \mathrm{T(G)} && \text{(by induction)} \\ &\leq c_0 F(u-l+1)\lg(u-l+1) + T(V) && \text{(monotonic } F\text{)} \end{aligned}$$

This is $O(F(u-l+1)\log(u-l+1))$, which is $O(F(n)\log n)$ at $T(V_1^n)$. ∎

The proof requires that sequentiality and bidirectionality not be present. If we have sequentiality then $T(V_l^u) = \max\left(T(V_l^{\lfloor (l+u)/2 \rfloor}), T(V_{\lceil (l+u+1)/2 \rceil}^u)\right) + T(G)$ does not hold because the computation of the V's can not proceed in parallel. If bidirectionality is present then we must have sequentiality. The proof goes through even if rootlock is present, but in such a case the theorem produces a weak result, since $F(n)$ would be large.

AD-A164 022 KNOWLEDGE-BASED TRANSFORMATIONAL SYNTHESIS OF EFFICIENT 2/3
STRUCTURES FOR CO. (U) KESTREL INST PALO ALTO CA
R M KING 30 SEP 85 KES.U.85.5 AFOSR-TR-85-1239
UNCLASSIFIED F49620-85-C-0015 F/G 9/2 NL





We must now briefly describe closures, which are the objects we intend to build using D&C. After we describe them, we show that use of closures does not produce slow parallel structures.

4.3 Description of Closures

Our solution to the problem of D&C formulations that do not meet our conditions of lacking bidirectionality, sequentiality and rootlock is based on the idea of passing a form of data called a *closure* up the tree, and therefore computing "big" closures (ones that perform a service for a large interval in the original problem vector) from "small" ones. A closure is a procedure or function definition together with an environment, i.e., a set of name/value pairs. When a closure is invoked, the procedure or function is invoked in the included environment as augmented by parameter binding. When processor A passes processor B a closure, A is said to be the closure's *host* and B the *recipient*.

A closure consists of a procedure, and bindings for some of the procedure's free variables. The procedure, in turn, consists of a piece of program and a binding list. The concept was first described in Church's λ-calculus [Chu51]. Closures are valued for their expressive power even on single-processor algorithms. They are elements primarily of dialects of LISP. See, for example, [Ste77], [Moo82], [XER83]. A similar concept, *actors*, is also found in other languages (see, for example, PLASMA in [SmH75].) Actors are also described as a method of expressing interprocessor communications concepts [AHe77]. We here explore a case in which our similar concepts aids a parallel structure computerized synthesis task.

The notation $\lambda X[F(X,Y)]$ denotes an abstraction of a function of n ($n \geq 0$) parameters from a function of $n+m$ ($m \geq 0$) parameters. X represents n bound variables; these variables are bound to the values of the n actual parameters when the function is called. Y represents m free variables, and the values used for them when $F(X,Y)$ is evaluated are determined by some of the context in which the abstracted function is evaluated.

We will use "$\lambda_X^Y[F(X,Y,Z)]$", where again X, Y and Z denote three groups of (respectively) n, m, and o variables, to denote a closure generating form (CGF). This is a piece of program text that, when evaluated, makes a closure that can be applied to as many parameters as there are elements of the X group. X is the group of bound variables and Y is the group of variables whose current values will be stored in the closure. Z is the group of free variables whose values at application time are to be used.

When a closure is applied, the X-values from the actual parameters in the application, the Y-values available at closure creation time and the Z-values at application time will be used. The Y group is constituted from the *closed variables*. We will use $\lambda_X^{Y=V}[F(X,Y,Z)]$, which would be created by the above CGF if $Y = V$ at the time the CGF is evaluated, to denote the closure in which $y_1 = v_1, y_2 = v_2, \ldots, y_m = v_m$.

An example of a CGF, taken from the synthesis of a parallel prefix summation unit, is $\lambda_z^{V_l,V_r,C_l,C_r}[(C_l(z) \parallel C_r(z+V_l))]$. The semantics of this is that when the form is evaluated its value is a closure, which is a functional object of one argument. The values of V_l, V_r, C_l, and C_r at the time the CGF is evaluated are "frozen" into the closure, and when that is applied the values are used. We get the closure $\lambda_z^{V_l=a,V_r=b,C_l=f,C_r=g}[(C_l(z) \parallel C_r(z+V_l))]$ when we evaluate the CGF in an environment in which $V_l = a$, $V_r = b$, $C_l = f$, and $C_r = g$. We see that C_l and C_r are, themselves, closures. The CGF calls for the creation of a closure that, when applied, binds four variables (one of which, V_r, does not appear in the form) to four saved values and z to the argument. It then applies z to one of the saved values and $z + V_l$ to the other simultaneously.

TRANSCONS will only generate a restricted subset of the possible closure generating forms. They will be of the form $C = \lambda_Z^{V_l,V_r,W}[G(Z,V_l,V_r,W)]$. V_l (resp. V_r) are the values, including closures which we will call i.e., C_l (resp. C_r), received from left (resp. right) children. W includes locally available values which may have been computed during previous upward communication phases, and G is of the form $G(V_l,V_r)(Z) = (C_l(G_l(Z,V_l,V_r,W)) \parallel$

$C_r(G_l(V_l, V_r \parallel G_r(Z, V_l, V_r, W))))$. (Here $\parallel$ is concurrent application, and G_l and G_r can impose side effects on W.)

We want to show that if the closure generating form has this property, i.e., that if a generated closure can be applied by doing a small amount of computation and applying other similar closures in turn, then application of the generated closure is fast. For convenience we will call this type of closure generating form a *tail applicable* form (by analogy with the term "tail recursion").

Theorem 4.3 *Suppose a closure is computed in the root of a balanced binary tree. That closure can contain closures whose hosts are its children. Those closures, in turn, can contain closures whose hosts are their children, etc. The leaves of the tree are closure hosts whose closures can be applied in time $O(1)$. Each leaf has a distinct index such that the set of indices of leaves is exactly a range of integers. The set of indices of every subtree's leaves is also a range of integers and is identified by the endpoints of that range, and the node heading the subtree whose leaves' indices are l through u is called n_l^u. Suppose all closures computed within the tree are tail applicable. If, in n_l^u, $\max(T(G), T(G_l), T(G_r)) = O(F(l-u+1))$, then $T(C_1^n) = O(F(n)\log n)$.*

Proof: Refer to $\max(T(G), T(G_l), T(G_r))$ in n_l^u as $T(n_l^u)$. $T(n_i^i) = O(1)$, so call it k. Say $T(G) \le c_0 F(u-l+1)$. We prove by induction that $T(n_l^u) \le c_0 F(u-l+1)\lg(u-l+1)+k$, where c_0 is the constant of $T(n_1^n) = O(F(n)\log n)$.

The base case is immediate.

If $l \neq u$ then

$$T(n_l^u) = \max\left(T(n_l^{\lfloor (l+u)/2 \rfloor}), T(n_{\lceil (l+u+1)/2 \rceil}^u)\right) + T(G) \quad \text{(definition, nonsequentiality, tree balance)}$$

$$\le c_0 F((u-l+1)/2)\lg((u-l+1)/2) + k + T(G) \quad \text{(by induction)}$$

$$\le c_0 F(u-l+1)\lg(u-l+1) + k \quad \text{(monotonic F)}$$

This is $O(F(u-l+1)\log(u-l+1))$, which is $O(F(n)\log n)$ at $T(n_1^n)$. ∎

Note the similarity between this proof and that of Theorem 4.2

Our technique will be to reformulate the problem from that of creating some new array that is a function of an existing array to that of creating a closure which, when invoked, will perform a given action on the leaves of a tree. This action is the creation of an *element* of the new array in each leaf. The original specification is transformed into a specification that declares the existence of a closure that, when invoked, will satisfy the original specification, followed by a specification that the new closure be invoked. In the synthesis process, the specification that a closure with certain properties exists is transformed into code that creates such a closure. This code has, as input, closures having the desired property in relation to subproblems of the problem.

Consider the process of combining two closures. The result will be a closure. In D&C with closures, it is normal to combine a pair of vectors, each containing computed values and closures, into a new vector with similar texture. We will consider the computation of a closure for the output vector, and the use of the resulting closures. We give informal reasoning to justify our assertion that closure generation is an effective technique for producing tree structures by D&C; we then prove formal versions of the informal assertions we make.

The combination operator can operate on values and closures from the input vectors to produce a new closure. The code before the closure generating form (CGF) may compute values that will be included in the closure, and the CGF itself will create an environment in which these values, as well as other values and some closures that were part of the input vectors, are saved. The combine operator is neither bidirectional nor sequential if we have reformulated the problem properly, and it avoids rootlock if pairwise combination of results from vectors produces new results not much larger than those that were combined, in an amount of time not much longer than the amount of time used to develop each

of the old results. For our purposes we would want these amounts to increase at most polylogarythmically in the length of the string incorporated in the results.

When the computation uses the computed closures, each host will have computed the closure it hosts by computing and storing some values, storing received closures from its children, and arranging that when the closure is applied, formal parameters (if any) be bound to actual parameters, a computation be performed to obtain actual parameters for its closures' incorporated closures, and such closures, in turn, be applied. The computations of a closure must meet similar conditions to those met by the combination process and described above. It should be noted, however, that the incorporated closures can be applied in parallel and that there is no need for the calling procedure to await completion of any applications.

We have exchanged the difficulty of reasoning about two-way data flow with the need to reason about closures. We feel that this is a good bargain because reasoning about closures only requires the addition of new axioms to a theorem prover's data base, while two-way data flow requires changes in the way we look at D&C. In Section 4.2 we showed that this change of view costs little speed, and in Section 4.5 we show that no expressive power is lost.

We conjecture that this technique can bring most $O(\log n)$ and $O(\log^2 n)$ tree parallel structures within the reach of a D&C-based synthesis method. We support this conjecture by several syntheses in the next Chapter . Since a tree-structured processor is inexpensive to manufacture compared to more highly interconnected machines and seems to be reasonably powerful, we feel that automatic tools that make use of this family of topologies easier would be an important contribution to the technology of synthesis of parallel structures.

In summary, the technique of computing closures from component closures is a technique which, together with D&C, provides the ability to synthesize a wide variety of tree structures with few of the technical problems that other synthesis methods might encounter

concerning reasoning about path lengths or the cardinality of sets of nodes. It allows us to do this and to still produce the $O(\log n)$ (or $O(\log^i n)$ for small i) parallel structures we expect from trees.

4.4 Transmission of Closures

To transmit a closure from one processor to another, it is *not* necessary to transmit the entire program and all of the environment values, provided that the host processor stands willing and able to perform the work. All that is necessary is that the transmitting processor send a token of some sort. The receiving processor can save the token and later use the closure by sending back the arguments, the token, and control information.

When this is done, the processor sending the closure (and willing to do the work) is called the closure's *host*, and the receiving processor (which has a license to use the closure) is called the *recipient*.

We say that a closure is *live* if there is a possibility that it will be invoked at a given time. A closure becomes live when it is sent and remains live until the recipient reaches a point in its procedure past which it can not invoke the closure. We will have more to say about issues concerning the liveness of closures during the remainder of this Section.

Closures can be efficiently implemented in a reasonable machine model. Internally, a closure can be implemented as a block of memory locations containing a "pointer" to the program fragment and a list of all closed variables and the corresponding values. A pointer to the block could be used as the token. When a closure is applied, the recipient can send the host a copy of the token, together with whatever other information is needed (primarily the argument(s)). The host can, using the received token, invoke the proper fragment together with the proper environment including the arguments bound to the parameters, by using the information contained in the closure and message.

A piece of program text (in the host processor) that creates a closure is called a *closure generating form* or CGF, and a piece of text (in the receiving processor) that invokes one is called a *closure invoking form* or CIF. The class of closures generated by one CGF is a *family*. An instance of the family of closures generated by a specific CGF named *C* will be called a *C instance* or an *instance from C*. Members of a family differ only in the environments, since the code will be the same.

The required data transmission can be reduced in cases where it is possible to infer various things about the use of a closure. For example, if it is known that only one instance from a given CGF is live at a time, the host needs not send the token, but only the name of the CGF. That name would not vary and can be "assembled into" the CIF. This can be true even if there can be several CGF instances for a given CGF, provided that the host knows in what order the recipient will use the closures it receives. If there is only one CGF in a processor, and only one instance of the closures that it generates can be live at one time, the token can vanish; the fact that the "receiving" processor wants to apply any closure is information enough! The closure has been completely swallowed up; information *only* travels from the recipient to the host, even though the synthesis was performed as if data flowed only in the other direction.

A further simplification, of interest for the problem of synthesizing parallel structures that will later be reduced to VLSI, is available. Suppose the following conditions are met: Applying a closure does not include changing state in the host processor. (In this case, for the application to be useful it must cause other applications in the host.) Assume also that there is only one live closure in a given family at any time. Assume further that the values used in that closure to call other closures hosted elsewhere can be computed, using only values available to the host, by means of combinatorial logic (the code fragment is loop-free and consists only of operators chosen from a library of integrateable operators).

In this case it is possible to perform the closure using only "combinatorial logic" in the host processor. Specifically, no register need be provided to hold the closure's parameter in the host processor. Instead, logic must be provided to map a signal representing an application of the closure to signal(s) representing application(s) of the subsequently called closure(s). Registers are provided to hold all of the values of the closure. An example taken from the Parallel Prefix structure (whose derivation sketch is in the next Chapter) will make this clear.

We have the code fragment to synthesize a closure, namely $\lambda_z^{C_l,C_r,v_l}[C_l(z) \parallel C_r(v_l+z)]$[1]. Here we will observe that there is only one outstanding instance of the closure at any time, because the variable in which the closure is stored in the recipient is not indexed. The closure does nothing more than apply other closures to a function of its argument. Furthermore, the computation performed on the argument is "easy" (and can be directly implemented in VLSI). We can therefore use the circuit of Figure 4.1:

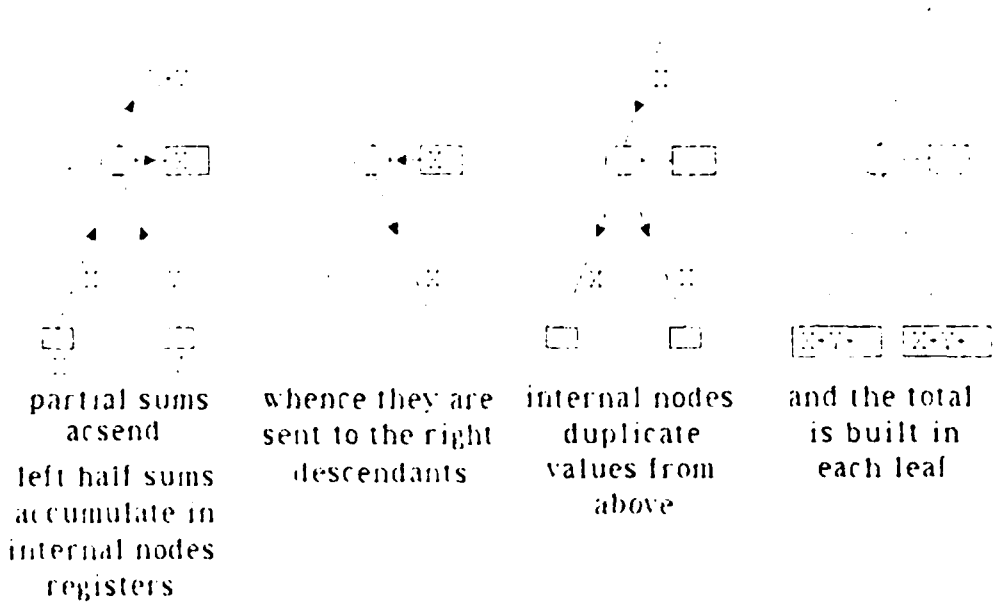


Figure 4.1: Simplified Parallel Prefix Internal Node

[1]For clarity, the exposition assumes that the prefix operation is addition, and that an addition module exists in our VLSI module library.

4.5 Arguments for the Completeness of Closures

In this Section we formalize the notions we use to argue that restricting communication to the upward direction in trees is a harmless restriction, not preventing the synthesis of tree parallel structures to meet any specification that could have been met absent this restriction, provided only that we also allow upward communication closures and that we not consider the application of a closure that was communicated upward to be a downward communication.

We need a formal definition of a tree parallel structure, and in order to do that we need to define some preliminary concepts.

Definition 4.2 *We define a* `binary tree` *in the usual manner. A tree either has a node called a* `leaf` *or a node called a* `root` *connected to the roots of two* `subtrees` *by edges which for these purposes we call* `wires`*. The root of a subtree is called an* `interior node`*. The connection from a root to one of its subtrees is called the* `left` *connection, and that to the other subtree is called the* `right` *connection. A root is called the* `parent` *of the two subtree roots it is connected to, and the connection from a node to its parent is called the* `parent` *connection.*

Note that each wire is known as the parent connection at one end and either the left or the right connection at the other. This defines an ordinary binary tree. We further state that there is an ordering of the leaves.

Definition 4.3 *The set of* **ancestors** *of a node is recursively defined as the union of the node's parent (if any) and the parent's ancestors. The set of* **descendants** *of a node is recursively defined as the union of the node's left child, right child and their descendants. The set of* `left descendants` *(resp.* `right descendants`*) of a node is the left resp. right node together with its descendants. A tree's leaves are* **ordered** *if the leaves are indexed by a totally ordered set and if the index of leaf A is less than that of leaf B iff there exists a node of which A is a left descendant and B is a right descendant.*

So far we have only described standard binary tree structures and names of interconnections. We would also like to describe a computation structure, which is a structure together with a set of computations and communications attached to each node. Each node has an associated program.

For our purposes, we will require trees to be *homogeneous*, meaning that one program will be run in all of the internal nodes, and another single program, with the leaf index as a parameter, will be run in all leaf nodes. Programs may do reasonable computations and may send and receive on the attached wires. We also require, however, that the structure be *singly buffered*, meaning that when a program tries to receive information over a given wire, it will do nothing else until the program of the node at the other end of the wire tries to send, and when a program tries to send on a wire twice without the other program having tried to receive, the sending program will do nothing else until the other program tries to receive. Programs may perform closure application with no regard to these restrictions, but the transmission of the closures must have obeyed these conditions. Programs may test whether a line has, or can accept, data and therefore avoid waiting if it can't. The situation where neither program at either end of the wire can send or receive is possible, but only for a bounded amount of time (assuming correct, terminating programming).

Now we can define the primary parallel structure of this portion of TRANSCONS.

Definition 4.4 *A* **tree parallel structure** *(tree structure) is a collection of processors together with programs that is a tree with numbered leaves, is homogeneous, and is singly buffered.*

We need a definition of a tree parallel structure with upward communication only:

Definition 4.5 *An* **upward tree parallel structure** *is a tree parallel structure in which no communication is specified from any parent to any child. Closure application is* not *regarded as communication in this context.*

This is a formal definition of the objects described by **TREES** declarations, and in the rest of this Section we will explore some of the implications of this definition. In particular, we are interested in an assertion that limiting communication to an upwards direction, but allowing closures, gives the same expressive power as allowing communication in both directions but not using closures.

First we need a lemma.

Lemma 4.4 *Suppose we have two processors A and B with two wires ab1 and ab2 from A to B. These wires obey the "singly buffered" condition above. It is possible to simulate those two wires with a single wire with no more than a constant factor speed loss.*

Proof: The wire is driven by commands of the form **read**(abi, x) which accepts information from the wire abi and stores it in x while making abi no longer have information to offer; **send**(abi, x), where $i = 1$ or 2 which waits for the wire to be receptive and then puts the contents of x on the wire which makes it unreceptive but gives it data that can be **read**; **readable**(abi) which returns **true** iff data are present (and can be **read**); and **sendable**(abi) which returns **true** if data can be sent on the wire. It is possible to replace forms according to the following table:

read(abi, x) becomes (in B)

    while undefined($vabi$) **do**
      check()
    $x \leftarrow vab1$
    $vab1 \leftarrow$ **undefined**

readable(abi) becomes

    defined($vabi$)

send(abi, x) becomes (in A)

    while defined($wabi$) **do**
      check()
    $wabi \leftarrow x$

sendable(abi) becomes

    undefined($wabi$)

Insert "*check*()" sufficiently often in the procedure running in both A and B to guarantee execution periodically, with a period short compared to the time it takes to communicate between processors. The *check*() call in A checks whether $wab1$ and $wab2$ are defined. If either is defined, say $vab1$, *check*() sends the pair $(1, wab1)$ over the wire and does $wab1 \leftarrow$ **undefined**. The check call in B is a finite state machine. In its initial state, it checks whether there is anything to read on the wire; if there is, it reads it. This should be a number i; the FSM enters a state S_i. If *check*() is in S_i, then it will check whether $vabi$ is empty; if and only if so, it will read the next object from the wire and enter the initial state.

Enumeration of the sequences of actions on the two wires, actual and simulated, serve to establish correctness. That there is only a constant-factor slowdown can be derived

from the fact that *check*() does a constant amount of work unless it waits, that it only waits if (and as long as) the simulated machine would have waited, and that it replace each communication with a constant number (two) of communications. ∎

Now we can prove a fundamental theorem about unidirectional communication in a tree.

Theorem 4.5 *Suppose we have a tree parallel structure $\mathcal{T}$ without transmission of closures. Then it is possible to perform the same computation that $\mathcal{T}$ performs on an upward tree parallel structure.*

Proof: Simulate a second wire from the left (resp. right) child to its parent per the previous theorem. Call that wire C_l (resp. C_r) where it impinges on the parent and C_p where it impinges on the child. In what follows we describe the treatment for left children; the treatment for right children is analogous. The parent may contain **send**(left, x) and **sendable**(left); the child may contain **read**(parent,x) and **readable**(parent). Each of these is replaced by new text as follows:

send(left, x) becomes (in the parent)

> **read**(C_l, C)
> $C(x)$

sendable(left) becomes

> **readable**(C_l)

read(parent, x) becomes (in the child)

> **while undefined**(v) **do**
> *check*()
> $x \leftarrow v$
> $v \leftarrow$ **undefined**

send$(C_p, \lambda_x^v[v \leftarrow z])$

readable(parent) becomes

defined(v)

check() is similar to that of the previous theorem. Additionally, the fragment

send$(C_p, \lambda_x^v[v \leftarrow z])$

must be prepended to the child's program and

read(C_l, C)
$C(x)$

appended to the parent's.

That this causes correct information to be seen in the recipient is evident from the observation that each closure is used to send exactly one value to the recipient, exactly that value is used as an argument to the closure as was previously being sent, and it is only used once (and immediately rendered undefined). That this causes the programs to "hang" at exactly the right times can be easily seen from the fact that there is a closure in (say) C_l exactly when the recipient would have been receptive, and there is a value in v exactly when there would have been a value available. ∎

The key point to note is that all downward communication is expressed as closure application. This suggests that it will be possible to express a problem that apparently can not be solved by D&C as the corresponding problem of creating, in the root, a closure that has a desired result when applied.

We have therefore shown that we do not surrender any expressive power when we limit tree declarations to upward communication.

4.6 Trees of Processors in TRANSCONS

Trees of processors can be used to efficiently implement many specifications because the tree is that topology with fixed arity and lowest connectivity that allows a distinguished node to have contact with all other nodes in $O(\log n)$ steps, which is clearly the best possible. TRANSCONS ([Kin83]) therefore has facilities for specifying, synthesizing and manipulating trees.

The description of a tree is specified in **TREE** declarations, described below. Before describing the syntax of a **TREE** declaration, we will describe some of the semantics we intend for it.

The trees we intend to address are used to shorten the longest path lengths within the collection of processors, and to balance the workload of a computation. There are problems amenable to a tree solution, portions of which are in some sense more important than others (for example Optimal Binary Search Trees), but in these problems there must be a specification of relative importance that has a size comparable to the size of a good specification of the solution. We will therefore model solutions to problems of this sort by building separate trees and **AGGREGATE**ing them. Each tree described in a single locution will be balanced.

Several principles govern the design of the tree system of TRANSCONS.

- All trees are as balanced as possible. (We use binary trees; extensions to trees of higher arity introduce no new principles.) *No flexibility in terms of shape is assumed, nor is any way provided for expressing shapes.*
- A tree specification must include a size, which can be any integer greater than one.

- The shapes of two trees of the same size are identical. That is, there is an isomorphism $\simeq$ between two trees of the same size that maps parents, left children and right children respectively into parents, left children and right children. There are "compile-time" constructs in the TRANSCONS language that allow for the specification of connections to the node that is $\simeq$ to a given node, or **AGGREGATION** between corresponding nodes of different trees. One way to achieve this identity of shape is to have a left-biased tree that is as balanced as possible. In other words, path lengths from root to leaves differ by at most one and if one such path is longer than a second the first path must be to the left of the second.

- The nodes of a tree are divided into three groups. They are the root, the internal nodes, and the leaves. The leaves are further distinguished by indices. References to any of these classes of tree nodes, either to attach procedure, to specify communication such as **HEARS**, or to **AGGREGATE** can be made. Tags are provided for a node to refer to a node of another tree that is $\simeq$ to it if the two trees are the same size. This allows nodes in $\simeq$-equivalence classes to be **AGGREGATED** or to **HEAR** each other. For this to work, values have to be declared properly. Note that a leaf has to offer instances of values that are **HEAR**d upward, and the root has to offer values that are **HEAR**d downward.

To support these stipulations we have the **TREE** data type. A tree is declared and its components laid out using the type facility of CHI. As an example, we will describe below a situation where there are two trees, T and U. Each is of size n. Each internal node of T passes a value to its children after having multiplied it by a value from the corresponding internal node of U. Each internal node of U adds values from its two children. The procedures at the leaves of T and U, respectively, are described by functions H and G, not interpreted here.

T istype TREE (i) size n

root HAS v	TALKS *leftson*	(SENDS v)
	TALKS *rightson*	(SENDS v)
	HEARS *source*	(USES *outside−value*)
	HEARS *U.root*	(USES *u−value*)
inter HAS v	TALKS *leftson*	(SENDS v)
	TALKS *rightson*	(*unskip*SENDS v)
	HEARS *parent*	(USES v)
	HEARS *U.inter*	(USES *u−value*)
leaf (i) HAS l_i	HEARS *parent*	(USES v)

U istype TREE SIZE n

root HAS u	TALKS *T.root*	(SENDS u)
	HEARS *leftson*	(USES v as *v.left*)
	HEARS *rightson*	(USES v as *v.right*)
inter HAS u	TALKS *T.inter*	(SENDS u *as* *u−value*)
HAS v	TALKS *parent*	(SENDS v)
	HEARS *leftson*	(USES v as *v.left*)
	HEARS *rightson*	(USES v as *v.right*)
leaf (i) HAS v	TALKS *parent*	(SENDS v)
	HEARS $some_i$	(USES A_i)

(in *T.root*)

$v \leftarrow outside{-}value \times u{-}value$

(in *T.inter*)

$v \leftarrow v \times u{-}value$

(in $T.leaf_i$)

$l_i \leftarrow H(v)$

(in *U.root*)

$v \leftarrow v.left + v.right$

(in *U.inter*)

$v \leftarrow v.left + v.right$

$u \leftarrow v$

(in $U.leaf_i$)

$v \leftarrow G(A_i)$

Note the **SENDS** u **AS** *u−value* locution. This causes a value to be known as u in the Intermediate node but to be known as *u−value* in the recipient.

This example displays three features of tree structures. The U tree has upward communication and uses v to name links on which information flows from the leaves to the root.

There is also cross communication using $u-value$. Finally, there is downward information flow in T again using the name v.

In the next Chapter we explore the syntheses of several tree structures in detail.

Chapter 5

Tree Structures Synthesis Examples and Closure Removal

In this Chapter, we will consider the broadcast problem, the prefix summation problem, and a part of one solution to the connected components problem that is amenable to tree solution.

Our motivation is to display the tree synthesis methods of TRANSCONS in some detail. We first show the use of D&C to synthesize some tree structures that include closures. We then exhibit closure removal techniques that convert these into structures that can be implemented in computation models that do not permit closures.

The division step requires some explanation. Our basic technique is to assert that there exists a closure whose action is to make true the required first order logic (FOL) predicate, and then to make the goal to compute this closure and to apply it. The next step is to assert that there is a solution to the problem, if there is a solution to appropriate subproblems. The problems are expressed as input/output behavior on vectors, and all problems and subproblems are concerned with (contiguous) subvectors of the problem instance vector. Different methods of performing the division step result in different tree structures.

Specifically, we have the hypothesis $\exists C_l^u, u'[\mathbf{action}(C_l^u()) = \forall i \in \{l \ldots u\}[P(i)]] \wedge C_l^u \equiv G(C_l^{u'}, C_{u'+1}^u)$. This isn't strong enough for us to synthesize a tree structure, because we have to know more about usable values of u' given l and u. If we have $\exists C_l^u \forall l < u' \leq$

$u[\mathbf{action}(C_l^u()) = \forall i \in \{l \ldots u\}[P(i)]] \wedge C_l^u \equiv G(C_l^{u'}, C_{u'+1}^u)$, then we can certainly make a balanced binary tree (or any other tree structure we choose); our choice of u' is unrestricted (except that it has to split the range into two nonempty pieces).

Other interesting possibilities include $\ldots[u' = l + 1 \wedge \mathbf{action} \ldots$ or $\ldots[u' = u - 1 \wedge$ **action**..., both of which would create trees that are as unbalanced as possible (identical to chains), and $\exists C_l^u, u'[u - l \leq 3(u' - l) \leq 2(u - l) \wedge \mathbf{action} \ldots$, which gives us a tree structure with logarithmic depth, if the u' can be found at compile time. These cases provide interesting future research but are beyond the scope of this thesis.

5.1 Broadcast

In the broadcast problem, value(s) known in a central location are distributed to many locations. The broadcast problem can be described formally as $\forall i[a_i' \leftarrow F(a_i, x)]$ or perhaps $\forall j[\forall i[a_{ij}' \leftarrow F(a_{ij}, x_j)]]$. One method of synthesizing solutions to this problem might be to recognize it as a distinct pattern and carry a synthesis rule that produces a broadcast tree when supplied an instance of a broadcast problem. Another solution is to produce a chain of processors as a bucket brigade to distribute the information, and then to successively split the chain in half. The problem with this solution is that the synthesis process is iterated a variable number of times. With the new mechanism of closure passing, it is possible to provide more general rules that handle broadcast problems as a special case without multiple reformulations.

Consider the application of D&C. We want to produce a closure that, when applied to x, performs $\forall i[a_i' \leftarrow F(a_i, x)]$. We hypothesize that to solve the problem for a whole subarray, we can solve the problem for each of two pieces of the subarray and combine the two solutions in some manner. Giving the names fl and fr to the closures for the left and right halves of the problem, and fw to one for solving the whole problem, we then show that

to combine closures $fl = \lambda x[\forall j \in r_1[a'_j \leftarrow F(a_j, x)]]$ and $fr = \lambda x[\forall j \in r_2[a'_j \leftarrow F(a_j, x)]]$ we have only to create $fw = \lambda_y^{fl,fr}[fl(y) \parallel fr(y)]$. We go through the following sequence:

$$\begin{aligned}
& \forall A, x \exists A'[\forall i \in [1 \ldots n][a'_i = F(a_i, x)]] \\
\Rightarrow\ & \exists C(x)[\mathbf{action}(C(x)) = \forall A, x[\forall i \in [1 \ldots n][a'_i = F(a_i, x)]]] && \text{(abstraction)} \\
\text{hypothesis:}\ \equiv\ & \exists C_l^u, \forall u', l < u' \leq u[\mathbf{action}(C_l^u(x)) = \forall A, x[\forall i \in [l \ldots u][a'_i = F(a_i, x)] && \text{(division)} \\
& \qquad \wedge C_l^u(x) \equiv G(C_l^{u'}(G_l(x)), C_{u'+1}^u(G_r(x)))]]
\end{aligned}$$

The abstraction step is the step of asserting that there is a function whose application makes true the FOL predicate that is being abstracted. The division step is the step of asserting that it is possible to build a closure that solves a large problem, given closures that solve subproblems (and possibly other data).

In this case it is possible to assert that any u', $l < u' \leq u$, meets the requirements, and that in this case a balanced binary tree solution to the problem will certainly work because the division $u' = u + \lceil \frac{u-l}{2} \rceil$ will provide one.

Setting $G(C_1(x), C_2(y)) \equiv C_1(x) \parallel C_2(y)$ ($\parallel$ is concurrent evaluation) and $G_l(x) = G_r(x) = x$ will cause the division hypothesis to be true.

It only remains to describe the procedure for handling a singleton array. This is the closure $\lambda_x^i[a'_i \leftarrow F(a_i, x)]$.

The computation of the top level closure is $O(\log n)$ where n is the size of the problem. This is clear from the theorems of the previous Chapter and from the observation that $T(G) = O(1)$. (G is creation of a closure enclosing two given closures.) Similarly, the time consumed by an application of the top closure will be $O(\log n)$ from the fact that $\max(T(G_l), T(G_r)) = O(1)$. ($G_l$ and G_r are identity operations.)

5.2 Parallel Prefix Summation

Prefix summation is a mapping of arrays onto arrays of the same size such that the i^{th} element of the output array is the sum of the first i elements of the input array. This generalizes to other reduction operations; it makes sense to talk about prefix product, prefix **and**, etc. The only restriction, shared with other reduction operations, is that the operation be associative.

Handmade parallel structures that solve the prefix summation problem have appeared in the literature. See, for example, [**LFi80**] and the more recent [**Fic83**]. Below we describe methods to synthesize such structures.

5.2.1 Overview

To use the closure technique on a given specification, reformulate the problem from something like $\forall X, \ldots \exists Y [P(X,Y)]$ to $\exists C [\forall X, \ldots \mathbf{action} C() = P(X,Y)]$. Heuristically, the problem is reformulated from that of satisfying a specific input/output specification to that of producing a closure wherein the I/O specification[1] will be satisfied when it is applied.

We will need to define "augmented prefix summation with augend z" as $\forall 1 \leq i \leq n [a'_i = z + \sum_{1 \leq j \leq i} a_j]$. We then say that the task is to deliver, to the root of the tree, a closure that will perform augmented prefix summation. To create a closure that will perform augmented prefix summation with augend z on a non-trivial vector, divide it into two halves, get such a closure from each half together with the grand total of the input values for that half, and invoke the left half's closure with z as an augend and the right half's with the left half's sum added to z. We deliver, to each node of the tree, closures that will perform augmented prefix summation on the vector comprising its leaves, together

[1] More precisely, the problem of satisfying the I/O specification that requires no input and produces that closure.

with the leaves' sum. Note that the closure delivered to each node's parent has to include the left subtree's sum, which is available now but won't be later. A more formal description follows.

Assume that a vector $A_{[l\ldots u]}$ is divided into $A_{[l\ldots u']}$ and $A_{[u'+1\ldots u]}$. Further assume that we are trying to compute $F(A_{[l\ldots u]})$ which we will denote F_l^u. Further assume that we want to have some effects, local to the array elements. We would therefore want to compute a closure, C_l^u, that would have the desired effect.

The generic combination operator for the values is $F_l^u = G(F_l^{u'}, F_{u'+1}^u, l, u, u')$ and it is a synthesis task to derive the properties of G. Similarly, $C_l^u = G(C_l^{u'}, C_{u'+1}^u, l, u, u')$. If the closure has an argument, the situation is slightly more complex; we have $C_l^u(z) = G(C_l^{u'}(G_l(z, F_l^{u'}, F_{u'+1}^u, l, u, u')), C_{u'+1}^u(G_r(z, F_l^{u'}, F_{u'+1}^u, l, u, u'), l, u, u'))$ where the F vectors are the values available to (and incorporated in) C_l^u. This general schema need only be used with specific combiners (i.e., G, G_l, etc.). As a simple example, prefix summation can be performed by this schema if $G \equiv (C_{left} \parallel C_{right})$ (where $\parallel$ is concurrent application), $G_l(z) = z$, and $G_r(z) = z + v_l$. v, in turn, is computed as $v_l + v_r$. Singleton v- and C-expressions are $C_i \equiv \lambda z\,[a_i' \leftarrow a_i + z]$ and $v_i = a_i$.

5.2.2 Derivation

In prefix summation, the specification to meet is $\forall i \in \{1\ldots n\}[a_i' \leftarrow \sum_{j\in\{1\ldots i\}} a_j]$. We will introduce the abbreviation $\sum_l^u \equiv \sum_{j\in\{l\ldots u\}} a_j$. This then becomes $\forall i \in \{1\ldots n\}[a_i' \leftarrow \sum_1^i]$. We change the specification to one requiring the computation of a closure which, when applied to no arguments, performs this action, together with the application of that closure.

$$\forall A \exists A' \left[\forall i \in \{1\ldots n\}\left[a_i' = \sum_1^i\right]\right]$$
$$\Rightarrow\ \exists C \forall A \left[\mathbf{action}(C()) = \forall i \in \{1\ldots n\}\left[a_i' = \sum_1^i\right]\right] \qquad \text{(abstraction)}$$

$$\text{hypothesis: } \equiv \exists C_l^u \forall A \left[\mathbf{action}(C_l^u()) = \forall i \in \{l \ldots u\} \left[a_i' = \sum_l^i \right] \right. \qquad \text{(division)}$$
$$\left. \wedge\, C_l^u \equiv G(C_l^{u'}(), C_{u'+1}^u()) \right]$$

But $\mathbf{action}(C_{u'+1}^u()) = \forall i \in \{u'+1 \ldots u\}[a_i' = \sum_{u'+1}^i]$ so it is impossible for G to do anything to $C_{u'+1}^u$ to make its action $\ldots a_i' = \sum_{u'+1}^i]$. $C_{u'+1}^u$ must have a parameter in order to allow G to provide enough information.

We modify the closures so instead of $\mathbf{action}(C_l^u()) \equiv \ldots$ we have $\mathbf{action}(C_l^u(z)) = \forall i \in \{l \ldots u\}[a_i' = H(\sum_l^i, z, l, i)]$. We do not yet know the properties of H.

We now have:

$$\mathbf{action}(C_l^u(z)) = \forall i \in \{l \ldots u\} \left[a_i' = H\left(\sum_l^i, z, l, i \right) \right]$$
$$\mathbf{action}(C_l^{u'}(z)) = \forall i \in \{l \ldots u'\} \left[a_i' = H\left(\sum_l^i, z, l, i \right) \right]$$
$$\mathbf{action}(C_{u'+1}^u(z)) = \forall i \in \{u'+1 \ldots u\} \left[a_i' = H\left(\sum_{u'+1}^i, z, l, i \right) \right]$$

So we observe

$$\begin{aligned}
&\mathbf{action}(C_l^u(z)) \\
&\equiv \left(\forall i \in \{l \ldots u\} \left[a_i' = H\left(\sum_l^i, z, l, i \right) \right] \right) && \text{(above)} \\
&\equiv \mathbf{action}\left(G(C_l^{u'}(G_l(z,l,i)), C_{u'+1}^u(G_r(z,l,i))) \right) && \text{(D\&C)} \\
&\equiv G\left(\left(\forall i \in \{l \ldots u'\} \left[a_i' = H\left(\sum_l^i, G_l(z,i,i), l, i \right) \right] \right), \right. \\
&\qquad \left. \left(\forall i \in \{u'+1 \ldots u\} \left[a_i' = H\left(\sum_{u'+1}^i, G_r(z,l,i), l, i \right) \right] \right) \right)(z) && \text{(expansion)} \\
&\equiv \forall i \in \{l \ldots u'\} \left[a_i' = H\left(\sum_l^i, z, l, i \right) \right] \wedge \forall i \in \{u'+1 \ldots u\} \left[a_i' = H\left(\sum_l^i, z, l, i \right) \right] && (\forall \text{ identity})
\end{aligned}$$

Assuming G merely generates a closure to produce application of both of its parameters, then $H(\sum_l^i, z, l, i) = H(\sum_l^i, G_l(z,l,i), l, i)$ and $H(\sum_l^i, z, l, i) = H(\sum_{u'+1}^i, G_r(z,l,i), l, i)$. The first solves as $z = G_l(z,l,i)$. The second needs a bit more attention. If we

represent $\sum_l^i$ as $\sum_l^{u'} + \sum_{u'+1}^i$, we learn that $H(\sum_l^i, z, l, i) = H(\sum_l^{u'} + \sum_{u'+1}^i, z, l, i) = H(\sum_{u'+1}^i, G_r(z,l,i), l, i)$, so $H(r+q, z, l, i) = H(q, G_r(z,l,i), l, i)$ where $r = \sum_l^{u'}$.

Letting $H = \lambda x, y, z, w[y + x]$ we get $G_r = \lambda x[r + x]$. This can lead to two problems. One is that $H(x, y, z, z) = x$ must be satisfied, so it must be possible to satisfy $y + x = x$ for some y. At this point, if there is no identity to the operation, we have to say $G_l(x, z, z) =$ ⟨a special value⟩, and $H = \lambda x, y, z, w[y \oplus x]$ where ⟨special value⟩ $\oplus\, x = x$. The other problem is that there isn't enough information around to compute G_r. We have to expand the problem again to bring about the availability of intermediate values for the intermediate closures. In this case we need $\sum_l^{u'}$. Instead of

$$\textbf{action}\,(C_l^u(z)) \equiv \textbf{action}\left(G\left(C_l^{u'}(G_l(z,l,i)), C_{u'+1}^u(G_r(z,l,i))\right)\right)$$

we want

$$v_l^u = J(v_l^{u'}, v_{u'+1}^u)$$

and

$$\textbf{action}(C_l^u(z)) \equiv \textbf{action}\left(G\left(C_l^{u'}(G_l(v_l^{u'}, v_{u'+1}^u, z)), C_{u'+1}^u(G_r(v_l^{u'}, v_{u'+1}^u, z)\right)\right).$$

Taking a more intuitive view for the moment, we observe that we want to compute a pair (v_l^u, C_l^u) in which $v_l^u = \sum_l^u$ and in which $\textbf{action}(C_l^u(z))$ is the computation of an *augmented prefix summation*, where $a_i' \leftarrow z + \sum_l^i$ instead of $a_i' \leftarrow \sum_l^i$.

We want $G_r(v_l^{u'}, v_{u'+1}^u, z) = z + \sum_l^{u'}$, so we must use $v_l^{u'} = \sum_l^{u'}$ or $v_l^u = \sum_l^u$.

We lack only one step to a complete solution. Initially we wanted to compute a closure which, when computed for that "sub-array" which is the whole array and applied to no argument, computes the prefix sum. We will get, instead, a pair of results. One of the results is a value, and the other is a closure which, when applied to *one* value, computes a generalization of the prefix sum. It remains to convert this back into a closure that can be applied to no arguments.

We have

$$\mathbf{action}(C_l^u(z)) = \forall i \in \{l \ldots u\} \left[a_i' = \sum_j^i + z \right]$$

and we want

$$\mathbf{action}(F'()) \equiv \forall i \in \{1 \ldots n\} \left[a_i' = \sum_1^i \right] \equiv \mathbf{action}(C_1^n(z)) = \forall i \in \{1 \ldots n\} \left[a_i' = \sum_1^i + z \right]$$

for some z. Clearly $z = 0$ works. The operation will always have an identity because the creation of H will require a new operation to be created if the basic operation lacks one.

5.2.3 Derivation Summary

We have taken all of the following steps:

- We transformed an *I/O specification* whose input and output were vectors into an I/O specification taking vectors into a closure;
- We have substituted the I/O specification into a general D&C scheme;
- We established that the subclosures need an argument to fill their role in the closure that is being computed and modified the specification to reflect this;
- We augmented the specification to compute another value that is needed to compute the argument that subclosures will need;
- We performed backwards inference to determine the I/O behavior of generic functions from the D&C formulation; and
- We returned to the original problem of computing a closure to solve the original problem in terms of the new specification that specifies a closure that is a generalization of the function desired.

In slightly more detail, the steps are as shown below:

$$\begin{aligned}
\textbf{action}(C()) = \quad & \forall 1 \le i \le n \left[a'_i = \sum_1^i\right] \\
\textbf{action}(C'_l()) = \quad & \forall l \le i \le u \left[a'_i = \sum_l^i\right] \\
\equiv \quad & \forall l \le i \le u' \left[a'_i = \sum_l^i\right] \wedge \forall u'+1 \le i \le u \left[a'_i = \sum_l^i\right] \\
\equiv \quad & \forall l \le i \le u' \left[a'_i = \sum_l^i\right] \\
& \wedge \forall u'+1 \le i \le u \left[a''_i = \sum_{u'}^i\right] \\
& \wedge \forall u'+1 \le i \le u \left[a'_i = \sum_l^{u'} + a''_i\right]
\end{aligned}$$

We must supply a new parameter:

$$\begin{aligned}
\textbf{action}(C(z_0)) =& \forall 1 \le i \le n \left[a' = H\left(\sum_1^n, z_0\right)\right] \\
\textbf{action}(C'_l(z)) \equiv& \textbf{action}(C'^{u'}_l(G_l(z)), C'^{u}_{u'+1}(G_r(z)))
\end{aligned}$$

$H(\sum_l^i, z, l, i) = H(\sum_l^{u'} + \sum_{u'+1}^i, z, l, i) \simeq H(\sum_{u'+1}^i, G_r(z), l, i)$, which works if $H(x,y,z,w) = x+y$ and $G_r(z) = \sum_l^{u'} + z$, but the latter requires having $\sum_l^{u'} + z$ available. We therefore further modify the problem by requiring the collection of another value.

$$\begin{aligned}
v_l^u &= \sum_l^u \\
&= J(v_l^{u'}, v_{u'+1}^u) \\
&= J\left(\sum_l^{u'}, \sum_{u'+1}^u\right)
\end{aligned}$$

The last observations we need (the base case) are:

$$\begin{aligned}
v_i^i &= \sum_i^i = a_i \\
C_i^i &= \lambda z \left[\forall i \le j \le i \left[a'_j = z + \sum_i^j\right]\right] = \lambda z[a'_i = z + a_i]
\end{aligned}$$

We therefore have $J(x,y) = x + y$ making $v_l^u = v_l^{u'} + v_{u'+1}^u$. $C_l^u(z)$ applies $C_l^{u'}$ to z, and $C_{u'+1}^u$ to $z + v_l^{u'}$. Creating new symbols for the values (v_l, v_r and v) and closures (C_l, C_r, and C) received from the subproblems and passed to the superproblem, we finally get the following:

$$\begin{aligned}
J(x,y) &\equiv x + y \\
v &= v_l + v_r \\
v.leaf_i &= a_i \\
G(C_l, C_r) &\equiv C_l(G_l(z)) \parallel C_r(G_r(z)) \\
G_l(z) &\equiv z \\
G_r(z) &\equiv z + v_l \\
C &= \lambda_z^{C_l, C_r, v_l}[G(C_l, C_r)] \\
C.leaf_i &= \lambda z[a_i' = z + a_i] \\
C.root &= \lambda()^{C_l, C_r, v_l}[G(C_l, C_r)(0)]
\end{aligned}$$

We have $G(C_l, C_r) \equiv C_l(G_l(z)) \parallel C_r(G_r(z))$. We would therefore have a synthesized **TREE** declaration to read, in part,

```
. . .
inter HAS C, v
   HEARS leftson    (USES C as C_l, USES v as v_l)
   HEARS rightson   (USES C as C_r, USES v as v_r)
   TALKS parent     (SENDS C, SENDS v)
. . .
```

and the program for the internal nodes to read, in part,

```
(in T.inter):
  C ← λ_z^{v_l,v_r,C_l,C_r}[C_l(G_l(z)) || C_r(G_r(z))]
    where G_l(z) = z
    where G_r(z) = z + v_l
  v ← v_l + v_r
```

This can be converted to a tree structure free of closures by simple rewrite rules to be described in the next Section.

5.3 Closure Reduction

The parallel structure that results from the parallel prefix synthesis contains closure generating forms and closure applications. Permitting this makes the D&C synthesis process work—but the resulting structure is not a desirable one because the multiprogramming model must be used. Closure generating forms (CGF's) are not part of the lower levels because they require the processors to be capable of efficient context switches. Even if the multiprogramming model is satisfactory, there is one other reason for developing technology to eliminate closures from parallel structures. Any direct implementation of closures requires two messages to be sent in each of two directions: the closure from the host to the recipient, and the application message from the recipient to the host. This is wasteful because *the closure carries no information beyond collation (matching recipient-to-host messages with corresponding host-resident data) and indication of readiness.*

Both collation and readiness information are redundant in some parallel structures such as the one we have just synthesized for parallel prefix. Even where collation is necessary we will see that it can be realized by sending data only from the recipient to the host. For all of these reasons we are motivated to provide transformation rules that remove closures from a specification, replacing them with equivalent transmissions of argument data and possibly collation data. We call the process of transforming a parallel structure that includes closures into an equivalent one that does not, *closure reduction.*

There are two cases to consider: either there can be more than one live CGF instance at once, or there can not. We will call the first type of CGF a *multiple* CGF and the second a *simple* CGF. It is only required that the recipient send the host collation information to apply an instance of a multiple CGF, not of a single one. We will see below that the same reduction methods can be used for the two cases, with the redundant transmission being removed by further processing of the parallel structure that results from closure reduction.

5.3.1 CGF Reduction Rules

We describe an example of a case in which a CGF would support multiple instances. A two dimensional array of numerical values will be available from an external source. This array will be described as having m rows and n columns in what follows. The information will be available a row at a time. Each row has some maximal elements and some non-maximal ones. The problem is to determine, for each column, the sum of all elements that are maximal in their row.

There is a simple, fast, and memory-efficient tree-structured solution to this problem. A processor is assigned per column, and the column processors are connected by a binary tree. As the rows of the array arrive, the structure initiates a maximization calculation to determine the maximal element(s) of the row and send back, to those columns' processors that had maximal elements, a copy of that element. Each processor sums the replies it receives. It is possible to pipeline, i.e., to initiate subsequent maximizations before the results return from the first.

We reformulate the problem to that of computing, for all i in range, $C_i = \lambda z[\forall j \in (1 \ldots n)[a_j \leftarrow \text{if } b_{ij} = z \text{ then } a_j + z \text{ else } a_j]]$ and $v_i = \max_{k \in (1 \ldots n)} b_{ik}$ together with the application $C_i(v_i)$ for all i in range. We will use a division step in which substrings' closures are both applied either to the value of the maximal columns if the substring contains at least one, or 0 if the substring contains no maximal columns. We eventually get the following parallel structure:

. . .

inter HAS $C_i,\ 1 \le i \le m$

HEARS *leftson* (**USES** $C_i,\ 1 \le i \le m$ **as** Cl_i, **USES** $v_i,\ 1 \le i \le m$ **as** vl_i)

HEARS *rightson* (**USES** $C_i,\ 1 \le i \le m$ **as** Cr_i, **USES** $v_i,\ 1 \le i \le m$ **as** vr_i)

TALKS *parent* (**SENDS** $C_i, v_i,\ 1 \le i \le m$)

. . .

(**in** $T.inter$) :

$\forall 1 \le i \le m$

$C_i \leftarrow F(vl_i, vr_i, Cl_i, Cr_i)$

$v_i \leftarrow G(vl_i, vr_i)$

. . .

define $F(va, vb, Ca, Cb)$

(**return** $\lambda_z^{va,vb,Ca,Cb}$[(**if** $z = va$ **then** $Ca(z)$ **else** $Ca(0)$ || **if** $z = va$ **then** $Ca(z)$ **else** $Ca(0)$)])

define $G(va, vb)$

(**return** $\max(va, vb)$)

. . .

(**in** $T.leaf_j,\ 1 \le j \le n$) :

$s_j \leftarrow 0$

$\forall 1 \le i \le m$

$v_i \leftarrow a_{ij}$

$C_i \leftarrow \lambda_z^{v_i}$[**if** $z = v_i$ **then** $s_j \leftarrow s_j + z$]

(**in** $T.root$) :

$\forall 1 \le i \le m$

$C_i \leftarrow F(vl_i, vr_i, Cl_i, Cr_i)$

$v_i \leftarrow G(vl_i, vr_i)$

$C_i(v_i)$

In the parallel structure above we see that TRANSCONS must make certain changes. The transmission $C_i \leftarrow$ ⟨CGF⟩ where C_i is a communication variable must be changed into a mapping (in an appropriate map variable) of the point $(i, \langle\text{CGF name}\rangle) \rightarrow V$ where V is the vector of relevant values that are enclosed. A closure application $C_i(z)$ must be changed into transmission of the triple $(i, \langle\text{CGF name}\rangle, z)$ to the host. The host must include a new process whose procedure is the procedure portion of the closure augmented to take as input the values sent according to the previous change. A new pool must be created

to accomodate transmissions of triples to the host, and the pool that carries closures to the recipient can be removed.

We also make the cosmetic change of expanding the function definition $G(va, vb)$ in line. This definition results from the method we use of inserting a D&C scheme instance first and determining the necessary properties of the included functions later, and it will not be inserted in later examples. This example becomes

. . .

inter HAS $C_i,\ 1 \le i \le m$
  HEARS *leftson* (**USES** $v_i,\ 1 \le i \le m$ **as** vl_i)
  HEARS *rightson* (**USES** $v_i,\ 1 \le i \le m$ **as** vr_i)
  TALKS *parent* (**SENDS** $v_i,\ 1 \le i \le m$)
  TALKS *leftson* (**SENDS** $wl_i,\ 1 \le i \le m$ **as** w_i)
  TALKS *rightson* (**SENDS** $wr_i,\ 1 \le i \le m$ **as** w_i)
  HEARS *parent* (**USES** $w_i,\ 1 \le i \le m$)

. . .

(**in** $T.inter$) :
  $\forall 1 \le i \le m$
    $M_i \leftarrow (\mathcal{A}, vl_i, vr_i)$; *this is a map assignment that creates a "closure"*
    $v_i \leftarrow \max(vl_i, vr_i)$
(**in** $T.inter$) :
  $\forall 1 \le i \le m$
    $F'(i, w_i)$; *this awaits "closure applications"*

. . .

define $F'(i, ww)$; *when a "closure application" arrives...*
  if $M_{i,1} = \mathcal{A}$ **then**
    let $z = ww,\ va = M_{i,2},\ vb = M_{i,3}$ **do**
      $wl_i \leftarrow$ (**if** $z = va$ **then** z **else** 0) ; *"apply"*
      $wr_i \leftarrow$ (**if** $z = vb$ **then** z **else** 0) ; *contained closures*
      $M_i \leftarrow$ **undefined**

. . .

(**in** $T.leaf_j,\ 1 \le j \le n$) :
  $s_j \leftarrow 0$
  $\forall 1 \le i \le m$
    $v_i \leftarrow a_{ij}$
    $M_i = (\mathcal{B}, v_i)$
  $\forall 1 \le i \le m$
    let $ww = w_i$ **do**
      if $M_{i,1} = \mathcal{B}$ **then**

let $z = ww_1$ **do if** $z = M_{i,2}$ **then** $s_j \leftarrow s_j + z$
$M_i \leftarrow$ **undefined**
(in $T.root$**)** :
$\forall 1 \leq i \leq m$
$M_i \leftarrow (vl_i, vr_i)$
$v_i \leftarrow \max(vl_i, vr_i)$
$F'(i, v_i)$

Here A is the name given to the CGF "$\lambda_z^{va,vb,Ca,Cb}$[(**if** $z = va$ **then** $Ca(z)$ **else** $Ca(0)$ || **if** $z = vb$ **then** $Cb(z)$ **else** $Cb(0)$)]" and B the name of "$\lambda_z^{v_i}$[**if** $z = v_i$ **then** $s_j \leftarrow s_j + z$]".

The enabling conditions of the transformation rule are that a CGF exists, and that a communication variable exists that must be assigned an object of type closure. The CGF is given an arbitrary name. Creation of a closure is replaced by creation of a tuple giving the name of the CGF and the values of those elements of the closure that occur free in the procedure of the CGF. Assignment of the closure to a communication variable is replaced by entering the indices and name (if necessary) of the variable into a map, M, that maps it into the tuple that represents the closure. We add to any processor that contains one or more CGF's a process that awaits communications, decides which CGF it was based on by using M sets up the environment also stored in M and performs the procedure. A closure application is replaced by a transmission of index and argument information to the host.

If a CGF is situated so only one instance can be live at a time (determined by data flow), further simplification is possible. The simplification process begins as above, but after the closure passing and application is reduced the index portion of the transmission resulting from the closure application can be determined by flow analysis and need not be included. It can also be determined that the map will never contain more than a single element—so map insertion can be replaced by assignment of a variable and map retrieval by reference to the variable.

Below we show the parallel structure for parallel prefix summation of n-element vectors before and after closure reduction. Observe that the possibility of multiple CGF instances does not arise because within any one processor a CGF is only used once per computation.

. . .
inter HAS C
HEARS *leftson* (**USES** C **as** Cl, **USES** v **as** vl)
HEARS *rightson* (**USES** C **as** Cr, **USES** v **as** vr)
TALKS *parent* (**SENDS** C, **SENDS** v)
. . .
(**in** $T.inter$) :
$C \leftarrow F(vl, vr, Cl, Cr)$
$v \leftarrow vl + vr$
. . .
define $F(va, vb, Ca, Cb)$
(**return** $\lambda_z^{va,vb,Ca,Cb}[(Ca(z) \parallel Cb(va + z))]$)
. . .
(**in** $T.leaf_j$, $1 \leq j \leq n$) :
$v \leftarrow a_j$
$C \leftarrow \lambda_z^{a_j,a'_j}[a'_j \leftarrow z + a_j]$
(**in** $T.root$) :
$C \leftarrow F(vl, vr, Cl, Cr)$
$C(0)$

The result of the closure removal process is

. . .
inter HAS C
HEARS *leftson* (**USES** v **as** vl **as** vr)
HEARS *rightson* (**USES** v **as** vr)
TALKS *parent* (**SENDS** v)
TALKS *leftson* (**SENDS** wl **as** w)
TALKS *rightson* (**SENDS** wr **as** w)
HEARS *parent* (**USES** w)
. . .
(**in** $T.inter$) :
$M \leftarrow (A, vl, vr)$
$v \leftarrow vl + vr$
(**in** $T.inter$) :

$F'((), w)$

. . .

define $F'(i, ww)$
  if $M_1 = \mathcal{A}$ **then**
    let $z = ww,\ va = M_2,\ vb = M_3$ **do**
      $wl \leftarrow z$
      $wr \leftarrow va + z$
      $M \leftarrow$ **undefined**

. . .

(**in** $T.leaf_j,\ 1 \leq j \leq n$) :
  $v \leftarrow a_j$
  $M \leftarrow (\mathcal{B}, a_j)$
  let $ww = w,$ **do**
    if $M_1 = \mathcal{B}$ **then**
      let $z = ww_1,\ aa = M_2$ **do** $a'_j \leftarrow z + aa$
      $M \leftarrow$ **undefined**
(**in** $T.root$) :
  $M \leftarrow (vl, vr)$
  $v \leftarrow vl + vr$
  $F'((), v)$

The fact that the index into the map can only take a single value and is therefore redundant is immediate here, because the index is a vector of length zero! In other cases, preexisting value flow techniques such as those of [Ken81], [SPn81] and [CRi81] would be used to establish this fact.

5.4 Connected Components

We now explore another specification that raises some additional issues about D&C to synthesize tree structures and about closure removal. In this structure use of a closure causes another closure to be sent, because the use of a closure adds an element to a set that is being built up and this can be done repeatedly. We describe a closure removal technique that copes with this complication, and we sketch two possible implementations. The problem, together with one of the implementations, is described in [HMS84].

The problem is to find the connected components of a graph, given an adjacency matrix (a matrix A in which a_{ij} = true iff node i is (directly) connected to node j in the graph). The adjacency matrix will be available for input one row at a time, and a solution is preferred that reads the rows at equal intervals.

In this Subsection we will derive a tree structure that solves part of the problem and meets certain worst case time constraints. The derived structure will operate while the rows of the adjacency matrix are read in.

Formally, we will assume that there exists a source of rows of the adjacency matrix that can provide one row at a time. Each column will be read by its own processor. Columns and rows have integers in the range $[1, 2, \ldots, n]$ as names. When column i's processor reads row j it receives the value true if there is a graph edge between i and j or false otherwise. The network we derive will then store the information in such a manner that it or some other network can identify connected components of the graph whose adjacency matrix was read.

The adjacency matrix contains $\Theta(n^2)$ bits, and any system capable of storing this amount of information must obviously occupy proportional area. We would like to perform filtration or reduction of the n^2 bits of information into $n \log^i n$ for constant i, in order to make a more compact implementation of a circuit possible.

The column processor nodes of the network must read elements of the rows of the adjacency matrix at such a time (in relation to the time other processors read their elements of the same row) that the network will not confuse elements of different rows of the matrix, and the net must build a representation of the (partial) connected components information in some useful manner. The representation should be compact and the computation should be fast.

First we will derive the structure up to one important implementation decision; then we will describe the two resulting parallel structures.

5.4.1 Derivation of a Tree Structure

In the connected components problem, we do not necessarily want to change the state of the leaves of the tree or develop a value at the root. Instead, we want to change some state so questions about connected components become easier to answer.

We will use the notation $CC(i)$ to denote the set of nodes in the same connected component as the node i. $CC'(N)$ is a predicate indicating whether all nodes of N, a set of nodes, are in a single connected component. Since the state of knowledge of the connected components of a graph can vary with time and, in a multiprocessor system, with location, we will later introduce other variants of the CC' predicate.

We will read the rows of the adjacency matrix one by one. After we have read all of the rows we will then engage in another computation not described here, to put $\mathbf{reduce}_{\min}\{j : j \in CC(i)\}$ in leaf i. In what follows, we will call the processing that takes place between the reading of consecutive rows of the matrix a *phase*.

There are several solutions to the connected components problem which we reject because they have certain undesirable features. One solution, for example, would be to have each node record the row numbers of all rows of the adjacency matrix in which it is mentioned. This would require a lot of storage. Another solution is to have each leaf, after each row, find $\mathbf{reduce}_{\min}\{j : j \in CC(i)\}$ so far. The problem with this solution is that the time between the reading of rows can vary over a wide range (see [LiV81]).

Our derivation requires a certain amount of invention. We will assume that the user provides this by defining several intermediate predicates and by providing some information. Two ideas are involved in our conception: the idea of a map to store the state of the connected components so far, and the idea that the map is limited.

We start with axioms about connected components:

$$CC'(\{e\})$$

$$CC'(A) \wedge CC'(B) \wedge A \bigcap B \neq \emptyset \Rightarrow CC'(A \bigcup B)$$

$$CC'(A) \wedge A' \subset A \Rightarrow CC'(A')$$

We observe that the following is true:

$$CC'(A) \wedge CC'(B) \wedge \exists a, b[a \in A \wedge b \in B \wedge CC'(\{a, b\})] \Rightarrow CC'(A \bigcup B)$$

First, we supply TRANSCONS with a divide-and-conquer formulation. In what follows, V will be a set of connected components, each of which is a set of graph nodes; W is a connected component or a subset of one; CC (CC', etc.) is a predicate indicating whether a certain one of its arguments is known to be contained in a connected component; and M is a mapping of nodes to nodes.

$$\forall V[\forall W \in V[CC'(W)]]$$
where
$$CC'(W) \equiv \; |W| \leq 1$$
$$\vee$$
$$\forall W_l, W_r$$
$$[\; W = W_l \uplus W_r$$
$$\Rightarrow \; CC'(W_l)$$
$$\wedge CC'(W_r)$$
$$\wedge \; (W_l \neq \emptyset \wedge W_r \neq \emptyset \Rightarrow \forall a \in W_l, b \in W_r[CC'(\{a,b\})])]$$

TRANSCONS can easily check that this meets the axioms, but the combination of the two halves by a pair of arbitrary elements, one from each half, constitutes a user-supplied invention.

The user must observe that the current state of CC' is represented by the choices of pairs of arbitrary elements, and introduces M to carry this information. Since M represents the state of knowledge of connected components, we will define a new binary predicate $CC(M,X)$ which denotes that the mapping M asserts that there exists a connected component C such that $X \subset C$. Taking a finite difference against the addition of a new set X that is known to be connected, we get:

$$\forall X, M \exists M'[\; CC(M',X) \wedge \forall W[CC(M,W) \Rightarrow CC(M',W)]$$
$$\wedge \forall a,b[\; \sim CC(M,\{a,b\})$$
$$\wedge \forall Y,Z[CC(M,\{a\} \bigcup Y) \wedge CC(M,\{b\} \bigcup Z)$$
$$\Rightarrow Y \bigcap X = \emptyset \vee Z \bigcap X = \emptyset]$$
$$\Rightarrow \sim CC(M',\{a,b\})]]$$
where
$$CC(M,W) \equiv \; |W| \leq 1$$
$$\vee$$
$$\forall W_l, W_r$$
$$[\; W = W_l \uplus W_r$$
$$\Rightarrow \; CC'(W_l)$$
$$\wedge CC'(W_r)$$
$$\wedge (\; W_l \neq \emptyset \wedge W_r \neq \emptyset$$
$$\Rightarrow \exists a \in W_l, b \in W_r[M(a,b)])]$$

The long conjunct on the second through fifth lines state simply that no connected components are implied by M' that aren't either implied by M or forced by X.

We invite the user to make another critical observation, namely that $\forall W[CC(M,W) \Rightarrow CC(M',W)]$ can be satisfied by $\forall a,b[M(a,b) \Rightarrow M'(a,b)]$. (S)he can further observe from the original axioms that $CC(\{a,b\} \wedge a \in A \wedge CC(\{b,c\}) \wedge c \in C \Rightarrow CC(A \bigcup C)$. We can thus liberalize the condition on M in CC as follows:

$$\forall X, M \exists M'[CC(M',X) \wedge M(a,b) \Rightarrow M'(a,b) \wedge \ldots]$$

where

$$\begin{aligned} CC(M,W) \equiv \quad & |W| \leq 1 \\ \vee \quad & \forall W_l, W_r \\ & [\quad W = W_l \uplus W_r \\ & \Rightarrow \quad CC'(W_l) \\ & \wedge CC'(W_r) \\ & \wedge (\quad W_l \neq \emptyset \wedge W_r \neq \emptyset \\ & \Rightarrow \exists a \in W_l, b[M(a,b) \wedge (b \in W_r \vee CC(M, \{b\} \bigcup W_r))])] \end{aligned}$$

This specification is suboptimal because it allows M to be multivalued. We will examine this solution in detail and see how it translates into a tree that maintains M in internal state. We will then see what can be done to improve this.

We therefore make a change in CC to express the fact that the divisions will always be made in the same manner, and that M need only be defined for one set of subsets of the universe. This change is the addition of a parameter, a subset of the universe (of nodes in the graph whose connected components we are seeking). Later we will repair another deficiency of this specification: that it allows M to be larger than we would like.

M will be made a ternary rather than a binary relation. $M(S,a,b)$ is true if a connects to b relative to S. The purpose of this is to limit the size of M.

The new parameter to CC ranges over particular subsets of the universe. It has two roles: it tells what version of M to use, and it restricts acceptable solutions to CC. $CC(S,M,X)$ is true only if there exist elements of $M(S',x,y)$, where $S' \subset S$, that show that X is connected. This is a stronger condition than the original $CC(M,W)$.

In addition to making M and CC relative to a given set S, we will introduce functions L and R such that $L(S) \uplus R(S) = S$. The motivation for this is that we are trying to establist a tree structure of sets and subsetsthat together comprise those sets; L and R are a Skolemization of the assertion that there is a way of dividing the universe, each of its two subsets, etc. that meets further conditions. The domain and range of L and R must meet $\text{domain}(L) = \text{domain}(R) = \text{range}(L) \bigcup \text{range}(R) \bigcup \{U\} - U$.

To formalize the new parameter of CC we write:

$$\forall V, X, M, W \in V \exists M' \forall a, b [CC(M,W) \wedge CC(M',X) \wedge M(S,a,b) \Rightarrow M'(S,a,b) \wedge \ldots]$$

where

$$CC(M,W) \equiv CC(U,M,W)$$

and

$$\begin{aligned} CC(S,M,W) \equiv\ & |W| \leq 1 \\ & \vee \\ & \forall L, R \\ & [\quad L(S) \uplus R(S) = S \\ & \wedge W_l = W \bigcap L(S) \\ & \wedge W_r = W \bigcap R(S) \\ & \Rightarrow\ CC(L(S), M, W_l) \\ & \wedge CC(R(S), M, W_r) \\ & \wedge (\ W_l \neq \emptyset \wedge W_r \neq \emptyset \\ & \quad \Rightarrow \exists a \in W_l, b \in W_r [M(S,a,b)])] \end{aligned}$$

In what follows we will use the locution P_S to denote "the processor responsible for the set S".

A closure is needed here to satisfy $CC(L(S),M,W_l)$ and $CC(R(S),M,W_r)$. This closure requires no arguments, because the processors for $L(S)$ resp. $R(S)$ have all of the

information they need to do their work. All elements of W_l resp. W_r are in the subtree headed by processor $L(S)$ resp. $R(S)$.

Application of the closure serves notice on descendant processors that they should be ready to add to their maps in a manner that comes from the fifth conjunct in the large expression. The need will be described below.

Now we can continue the synthesis process by applying transformation rules to **satisfy**$(\forall V, X, M, W \in V \exists M' \forall a, b[CC(M, W) \wedge CC(M', X) \wedge M(S, a, b) \Rightarrow M'(S, a, b)])$. We soon find ourselves transforming **satisfy**$(M'(S, a', b'))$.

Suppose we add an additional condition, $M(S, a, b) \wedge M(S, a, c) \Rightarrow b = c$. After we have replaced occurrences of M by occurrences of M' (as a constraint propagator would do when analyzing "$CC(S, M', W)$") and imposed this condition, we get the following:

$$\begin{array}{l} \cdot \quad \cdot \quad \cdot \\ \wedge (\ W_l \neq \emptyset \wedge W_r \neq \emptyset \\ \quad \Rightarrow \exists a \in W_l, b \in W_r[M'(S, a, b) \wedge \forall c[M'(S, a, c) \Rightarrow c = b]]) \\ \cdot \quad \cdot \quad \cdot \end{array}$$

We can not satisfy this last clause (the implicand) when $\exists c \notin W_r[M'(S, a, c)]$ because this conflicts with $M(S, a, b) \Rightarrow M'(S, a, b)$. $M'(S, a, c)$ is required by $M(S, a, c)$ and forbidden by the requirement that $\exists c' \in W_r[M'(s, a, c')]$.

However, we have $M(S, a, c) \Rightarrow CC(S, \{a, c\})$ and $CC(R(S), \{c\} \bigcup W_r) \wedge CC(S, \{a, c\}) \Rightarrow CC(S, \{a\} \bigcup W_r)$.

We therefore use $\vee$ to expose the fact that there are alternatives:

$$\wedge\Big(\ W_l \neq \emptyset \wedge W_r \neq \emptyset$$
$$\Rightarrow \exists a \in W_l, b \in W_r[\ \ (M'(S,a,b) \wedge \nexists c \neq b[M(S,a,c)]$$
$$\vee \exists c[M'(S,a,c) \wedge CC(S,M',\{c\} \bigcup W_r)])]\Big)$$

As it is known that $M(S,a,x)$ can only be asserted by the above, an inductive proof is available that $c \in R(S)$. This can therefore be replaced by

$$\wedge\Big(\ W_l \neq \emptyset \wedge W_r \neq \emptyset$$
$$\Rightarrow \exists a \in W_l, b \in W_r[\ \ (M'(S,a,b) \wedge \nexists c \neq b[M(S,a,c)]$$
$$\vee \exists c[M'(S,a,c) \wedge CC(R(S),M',\{c\} \bigcup W_r)])]\Big)$$

This gives two alternative ways to satisfy the specification. We can satisfy $M'(S,a,b)$ if $M(S,a,b) \vee \nexists_c[M(S,a,c)]$. satisfying the other disjunct is harder than this because it requires satisfaction of a predicate containing $R(S)$, so we prefer the first disjunct when it can be satisfyed. If we can't use the first disjunct, then we know $\exists c[M(S,a,c)]$ so we have only to satisfy $CC(R(S),M',\{c\} \bigcup W_r)$ for that c. This leads to:

$$\textbf{satisfy}\Big(\exists a \in\ W_l, b \in W_r$$
$$[(\quad M'(S,a,b) \wedge \nexists c \neq b[M(S,a,c)]$$
$$\vee \exists c[M'(S,a,c) \wedge CC(R(S),M',\{c\} \bigcup\ W_r)])]\Big)$$

$\rightarrow$

bind a **to arb**(W_l), b **to arb**(W_r) **in**
if $M'(S,a,b) \vee \nexists_c[M(S,a,c)]$ **then satisfy**$(M'(S,a,b))$
else satisfy$(M(S,a,c) \Rightarrow CC(R(S),M',\{c\} \bigcup W_r))$

5.4.1.1 Closure Requirements

In order to be able to satisfy $CC(R(S), M', \{c\} \bigcup W_r)$ we will need a closure. This closure requires an argument because only P_S knows c. Since we have $CC(R(S), M, W_r)$ we will need a closure $CC(R(S), M, W_r) \Rightarrow \mathbf{action}(C_r(z)) = CC(R(S), M', \{c\} \bigcup W_r)$.

Expanding $CC(\dots)$ and renaming the variables with a prime (i.e., v becomes v'), we need W_l', W_r', $CC(L(S), M', W_l')$, $CC(R(S), M', W_r')$, and $\Big(W_l' \neq \emptyset \wedge \dots \vee \exists c[\dots \wedge CC(R(S), M', \{c\} \bigcup W_r')]\Big)$. Since we know that $W_r' = W_r \vee W_r' = \{x\} \bigcup W_r$ for some $x \in R(S)$ and similarly for W_l', we see that to establish $CC(S, M', \{c\} \bigcup W)$ we establish (and need a closure to so establish) $CC(L(S), M', \{c\} \bigcup W_l)$ (or $\dots R(S) \dots W_r$).

Each node P_S applies C_l once or not at all, and C_r zero, one or two times. We therefore need two features in order to satisfy our requirement that each closure be applied exactly once: we need to have a distinguishable null message to apply each closure to when a node knows that it will not be needing it, and we need to have the application of a closure cause the host to send another closure that has the necessary capability. We index the communication variable to allow successive closures to be distinguished.

There is an important fact that must be noted in this case. Because one result of using a closure is the generation of a similar one, a CGF for a closure must be included in the procedure attached to that CGF. This requires the procedure to call itself recursively.

The final version of the CGF that adds a 1 element to P_S's connected component is:

$CC\text{-}add\text{-}element(z) \equiv$
  case
    $z = \mathbf{nil}$:
      $C_l(\mathbf{nil}) \parallel C_r(\mathbf{nil})$
    $z \in L(S) \wedge W_l \neq \emptyset$:
      $W_l \leftarrow z \bigcup W_l$
      $C_l(z) \parallel C \leftarrow \lambda_z^{W_l,W_r,C_l',C_r}[CC\text{-}add\text{-}element(z)]$
    $z \in R(S) \wedge W_r \neq \emptyset$:
      $W_r \leftarrow z \bigcup W_r$
      $C_r(z) \parallel C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r'}[CC\text{-}add\text{-}element(z)]$
    $z \in L(S) \wedge W_l = \emptyset \wedge W_r = emptyset$:
      $W_l \leftarrow z \bigcup W_l$
      $C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r}[CC\text{-}add\text{-}element(z)]$
    $z \in R(S) \wedge W_l = \emptyset \wedge W_r = emptyset$:
      $W_r \leftarrow z \bigcup W_r$
      $C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r}[CC\text{-}add\text{-}element(z)]$
    $z \in L(S) \wedge W_l = \emptyset \wedge W_r \neq \emptyset$:
      $W_l \leftarrow z \bigcup W_l$
      $Map\text{-}add\text{-}or\text{-}call\text{-}Cr()$
      $C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r}[CC\text{-}add\text{-}element(z)]$
    $z \in R(S) \wedge W_l \neq \emptyset \wedge W_r = \emptyset$:
      $W_r \leftarrow z \bigcup W_r$
      $Map\text{-}add\text{-}or\text{-}call\text{-}Cr()$
      $C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r}[CC\text{-}add\text{-}element(z)]$

$Map\text{-}add\text{-}or\text{-}call\text{-}Cr() \equiv$
  let $a = \mathbf{arb}\ W_l,\ b = \mathbf{arb}\ W_r$ **do**
    if $M(S,a,b) \vee \exists c[M(S,a,c)]$
      then $M'(S,a,b)$
          $C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r}[CC\text{-}add\text{-}element(z)]$
      else let $c = M(S,a)$ **do** $C_r(c)$
                $C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r'}[CC\text{-}add\text{-}element(z)]$

$C \leftarrow \lambda_z^{W_l,W_r,C_l,C_r}[CC\text{-}add\text{-}element(z)]$

We will now explore the issues involved in transforming this specification containing closures into an equivalent one with downward communication.

5.4.1.2 Closure Removal Issues

Only *CC−add−element* can create a closure *C*—there is only one call to this routine outside of itself, and it is tail recursive. This implies that there is no way two CGF-instances can be live at once, allowing downward communication to consist of only the argument of the closure application.

As in the previous case we eliminate the closure by replacing the application with a setting of a communication variable and making the body of the CGF a piece of code that awaits such settings. The tail recursion that will occur unless the value of the communication variable is nil, is replaced by looping which tests for that terminating condition. The resulting code with closures eliminated is displayed below.

```
CC−add−element(z) ≡
  case
    z = nil :
      z_l ← nil;  z_r ← nil
    z ∈ L(S) ∧ W_l ≠ ∅ :
      W_l ← z ⋃ W_l
      z_l ← z
    z ∈ R(S) ∧ W_r ≠ ∅ :
      W_r ← z ⋃ W_r
      z_r ← z
    z ∈ L(S) ∧ W_l = ∅ ∧ W_r = emptyset :
      W_l ← z ⋃ W_l
    z ∈ R(S) ∧ W_l = ∅ ∧ W_r = emptyset :
      W_r ← z ⋃ W_r
    z ∈ L(S) ∧ W_l = ∅ ∧ W_r ≠ ∅ :
      W_l ← z ⋃ W_l
      Map−add−or−call−Cr()
    z ∈ R(S) ∧ W_l ≠ ∅ ∧ W_r = ∅ :
      W_r ← z ⋃ W_r
      Map−add−or−call−Cr()

Map−add−or−call−Cr() ≡
  let a = arb W_l,  b = arb W_r do
    if M(S,a,b) ∨ ∄c[M(S,a,c)]
```

then $M'(S,a,b)$
else let $c = M(S,a)$ **do** $z_r \leftarrow c$

(**in** *T.inter*) :
until $zz = \mathbf{nil}$ *do*
await defined(z)
$zz \leftarrow z$
CC−add−element(zz)

This completes the synthesis of the downward communication portion of a parallel structure for collecting connected component information from a series of rows of an adjacency matrix.

5.4.2 Alternative Data Structures

It is now necessary to consider the options for storing M. The type of M is $T \times U \rightarrow U$, where U is the set of nodes in the graph whose connected components are being determined, and T is a set of sets such that $U \in T \wedge (S \in T \wedge |S| > 1 \Rightarrow R(S) \in T \wedge L(S) \in T)$. The genesis of T is such that each intermediate node plus the root of the tree has as its set of leaves some element of T if each element of U is represented by a leaf.

Because of the type of M, we have four simple options to represent the mapping: We can represent it in one processor's memory, in the memory of one processor per element of T, in one processor per element of U, or in one processor per element of $T \times U$. The first possibility would lack concurrency and the last would require too many processors. The remaining possibilities include using interior nodes of the tree (corresponding to elements of T) or leaves (corresponding to elements of U) as the repository for information about parts of M.

Inspection of the specification yields the information that the tree node representing a set S must be able to answer questions of the form $\exists c[M(S,a,c) \wedge c \neq b]$ and

find c **suchthat** $M(S,a,c)$, and must be able to **satisfy**$(M(S,a,b))$. This requires either keeping $M(S,x,y)$ in S's node or providing that node with appropriate closures.

That node must also be able to **satisfy**$(CC(L(S),M',W_l))$ to **satisfy**$(CC(R(S),M',W_r))$, and to **satisfy**$(CC(R(S),M',c \bigcup W_r))$ given $c \in R(S) \wedge CC(R(S),M',W_r)$. This requires another handful of closures.

Since closures to **satisfy**$(CC(L(S),M',W_l))$ and **satisfy**$(CC(R(S),M',W_r))$ would require only information available below $L(S)$ and $R(S)$ respectively, and since there is no control flow path by which the need to **satisfy** these two predicates would be evaded, we observe that each interior node requires $a =$ **arb** W_l, $b =$ **arb** W_r, and the closure $\lambda_z^{M',a,b}$[**satisfy**$(M'(R(S),a,z))$].

We are building a map that maps at most one leaf of the right subtree to each leaf of the left subtree. As described, the map is stored in the node that has the appropriate subtrees. However, other alternatives are possible.

There are three natural places to store the assertion $M(S,a,b)$. They are the node whose subtree's leaves are S, leaf a, and leaf b. If the information is stored in S, there must be one cell for each leaf of the left subtree, and if the information is stored in a then there must be one cell for each ancestor representing S. If the information is stored in b, we have no limit (beyond the size of the problem) for the amount of storage that must be provided in b. We therefore reject this alternative.

Storing M in the node heading S minimizes communication (information is where it is used) making the algorithm take $O(\log n)$ steps. These steps are not constant-time steps, however, because they require access to a random access memory whose size is $O(n)$, itself an $O(\log n)$ operation. The algorithm therefore has an $O(\log^2 n)$ running time. It should

be noted, however, that in current technology the constant factor is very small compared to constant factors on $\log n$ terms until the problem instance becomes very large.

The result could be transformed to place the fact of $M(S, a, b)$ in a. This would result in a different algorithm, one that requires the leaves to supply closures to access and modify the map.

There is an interesting problem here. We would prefer that the leaves not have to know about elements of T. It would therefore be necessary to have the M table within each leaf organized in a certain order and to have use made of this information in that fixed order. This requires that a "flame front" of subtree handling be arranged such that initially, the root is the tree for which you are trying to associate pairs of elements, and on succeeding subphases, the level at which we are trying to match descends. This algorithm has an $O(\log^2 n)$ execution time because there are $\log n$ subphases, each of which is $O(\log n)$.

We prefer the former data structure, in which $M(S, a, b)$ is represented in S, because the issue described in the previous paragraph does not arise. That structure will always be available to us unless the size of a change to M is proportional to the size of S, and this can not be because the combination step of the divide and conquer scheme must be fast for the specification to parallelize well in a tree structure.

Below we describe in detail the algorithm that results from this decision, followed by that which results from storing the map in the leaves and a brief description of that synthesis path.

5.4.3 Results of Storing the Map in Internal Nodes

In this structure we have a cell in each subtree root (i.e., each internal node) for each of that subtree's leaves. This structure requires $n \log n$ cells, one for each leaf/ancestor pair, and it should lay out nicely in VLSI because the bigger nodes are closer to the root of the tree. Each cell must accommodate one of n vales, requiring $\log n$ bits. This imposes a total memory requirement of $n \log^2 n$ bits. Each internal node contains a map which maps names of leaves of the left subtree into either nil or names of leaves of the right subtree.

The overall view of the algorithm is as follows:

Each leaf sends its parent its name if its active, or nil. Each intermediate or root node sends its parent either the name of any active node it receives from its children, or nil if it receives nil from both children. If it receives two names it chooses arbitrarily. Each intermediate node also remembers what it received from its children.

In addition, suppose it receives a name from both children. There are two cases: If the name from the left node maps (in the node's internal mapping from leaves to values) into nil, make it map into the name from the right node and do nothing else. If it maps into (say) i, send awaken i to the right child and do nothing else.

If an intermediate node receives an awaken i node from its parent, it checks to see whether i is in its right or left subtree. It also checks to see what it has received before.

If a node receives an awaken i message and has already received a name from i's subtree it sends awaken i message to the appropriate child. If it hasn't received one it considers itself to have so received. (This can have one of three effects: modification of reaction to further awaken messages, lookup of i in the local mapping if i belongs in the left subtree, or lookup of a previously received name in the local mapping if i was in the right subtree and that previous name was in the left. If a lookup is performed we then either extend the mapping or create a new awaken message.)

The root sends its children a termination message when it's done. Intermediate nodes relay such messages. Each leaf reads the next line of the adjacency matrix when it receives this termination, and starts a new cycle.

The "wrapup", where each leaf gets the name of a representative of its connected component, is also faster under this arrangement. The root sends its right child its correspondences one by one, followed by "end". When a node receives $a \rightarrow b$ it replaces $b \rightarrow c$ (if it has one) by $a \rightarrow c$. This is not done for $b \rightarrow$ nil. Intermediate nodes also relay correspondences received from parents. When an intermediate node receives "end" from its parent, it dumps its own correspondences as they now stand and then sends its own "end". A leaf node initializes a cell to its own name and a cell named b changes this value to a if it receives $a \rightarrow b$. A leaf node knows it has the right value when it sees "end".

To derive this structure we make a different decision when creating the closures required by the synthesis. Rather than assuming that P_S needs no closure to **satisfy**$(M(S,a,b))$ or to test $\exists c[M(S,a,c)]$ we assume that closures necessary for either of these functions are available from $P_{L(S)}$, and ultimately from P_a. By a series of steps similar to the ones taken to synthesize the previous algorithm, we obtain a structure in which each leaf has a cell for each of its ancestors. This structure is described in [HMS84].

In this structure there are potentially $\Theta(\log^2 n)$ communications, because there are potentially $\log n$ phases in which it is learned that a left subset must link with a right subset, and each such phase requires $\Theta(\log n)$ communications to operate on the map data.

The parallel structure is (informally) as follows:

There is a balanced binary tree of processors where each leaf of the tree correspond to a node of the graph. For simplicity of exposition we will write the following as if the

leaves *were* rather than "corresponded to" the nodes. For simplicity we will assume that the entire adjacency matrix is supplied, rather than only a triangular matrix.

The leaf nodes build approximations to the answer as the algorithm grinds on. Each leaf node has one memory cell for each ancestor. Consider the memory cell for ancestor a in leaf l_i. It is initialized to the distinguished value **nil**, and during the course of the algorithm it will come to contain some j such that the least common ancestor (LCA) of j and i is a, and i and j are known to be in the same connected component, provided that some such j exists.

The algorithm works as follows: A leaf is called *active* if its bit is set in the current row of the adjacency matrix. After a row is read in, information is passed upward so each node can determine whether both of its subtrees contain active leaves, and what the highest and lowest active leaves are for such nodes. Information is then passed downward so each internal (or root) node can determine whether it is the top such node. That node sends a message to those two extreme nodes informing them of each other's identity.

The following cycle is repeated:

TU computes spans, TD distributes span information and keeps track of the topness of nodes.

TU **istype TREE** (i), $i \in [1,\ldots,n]$ **size** n
  root HAS *minact*, *maxact*, *topp*, *listop*, *ristop*
    HEARS *leftson* (**USES** *upmin*)
    HEARS *rightson* (**USES** *upmax*)
    TALKS *leftson* (**SENDS** *Listop*)
    TALKS *rightson* (**SENDS** *ristop*)
  inter HAS *minact*, *maxact*, *topp*, *listop*, *ristop*
    HEARS *leftson* (**USES** *upmin*)
    HEARS *rightson* (**USES** *upmax*)
    TALKS *leftson* (**SENDS** *listop*)
    TALKS *rightson* (**SENDS** *ristop*)
    TALKS *parent* (**SENDS** *upmin*)
    (**SENDS** *upmax*)
  leaf HAS $active_i$, $ccmate_{ij}, j \in$ **ancestors**
    HEARS *INPUT* (**USES** $adj_{ij}, j \in [1,\ldots,n]$)
    TALKS *parent* (**SENDS** *upmin*)
    (**SENDS** *upmax*)

(*in* $TU.leaf_i$)
$\forall j \in$ **ancestors**
  $ccmate_{ij} \leftarrow$ **nil**
$\forall j \in (1,\ldots,n)$
  $temp \leftarrow a_{ij}$
  $upmin \leftarrow upmax \leftarrow$ **if** *temp* **then** i **else nil**
  $dmin \leftarrow downmin$
  $dmax \leftarrow downmax$
  $other \leftarrow$ **nil**
  $pivot \leftarrow pivot$
  if $dmin = i$ **then** $other \leftarrow dmax$
  if $dmax = i$ **then** $other \leftarrow dmin$
  if $other \neq$ **nil then**
    if $ccmate_{i,pivot} =$ **nil**
      then $awaken \leftarrow$ **nil**; $ccmate_{i,pivot} \leftarrow other$
      else $awaken \leftarrow ccmate_{i,pivot}$

(*in* *TU.inter*)
;; *first establish my status*
$(lrangel, lrangeh) \leftarrow lrange$
$(rrangel, rrangeh) \leftarrow rrange$
$range \leftarrow (\min(lrangel, lrangeh), \max(rrangel, rrangeh))$
$livep \leftarrow range_1 \wedge range_2$

```
;; This is a once - per - minor - phase activity
  while dstatus ≠ 'dead


(in TU.root)
  (lrangel, lrangeh) ← lrange
  (rrangel, rrangeh) ← rrange
  range ← (min(lrangel, lrangeh), max(rrangel, rrangeh))
  livep ← range1 ∧ range2
  while dstatus ≠ 'dead


(in TD.inter),
  if pstatus ∈ {'live, 'top}
       then status ← 'live
              range ← prange
        elseif livep then status ← 'top
                            range ← range
        else status ← 'dead
  while status ≠ 'dead


(in TD.root)
        if livep then status ← 'top
                          range ← range
                  else status ← 'dead
```

In each minor phase the leaves send up awakening information and get back a packet of information very similar to the one they received in the beginning.

Each leaf, when it dies by finding that the node just above it is dead, sends up an "init" message. When every node has done so, the root broadcasts its own form of "init" and the leaves can read from the I/O processor that contains the next row of the adjacency matrix.

Here we describe the *overall* behavior of the algorithm, considering the parallel structure to be a single entity that can do things sequentially. To actually have this effect, there are synchronization problems, and below we describe a node's eye view of the situation, including the work that each node has to do to coordinate with its neighbors.

Initialize: Have each node read in its element of the adjacency matrix. Those nodes reading a "1" in the adjacency matrix turn themselves on, as does the node whose index corresponds to that of the row of the matrix. Mark the root as the "focus".

Survey: Every leaf sends information telling whether it is awake. Using this information, the internal nodes below a focus find out which of them has awake descendants in each of the two trees ("has two active subtrees"). This is a straightforward "up" problem.

New root: The highest node with two active subtrees is determined. This is the LCA of active leaves. It becomes the new focus, nodes between it and leaves become "active", and nodes above it but below and including the old focus become "dead".

Tournament: Select an arbitrary active leaf node in each of each focus's two subtrees. Report the identities of the two leaves to their focus. Simultaneously report the identity of the focus and of the other leaf to each of the two leaves.

Lookup: The leaves contain a variable mapping mapping their ancestors into a leaf index or the distinguished value nil. The leaves look up the focus in this mapping. If it is nil, they store the other leaf's identity. If the left leaf's value is not nil, report the value to its focus.

New awakening: If its left tree reports a leaf ID per Lookup, a focus sends a message to that leaf commanding it to awaken.

Refocus: Each focus sends a message to those of its children that are not leaves telling them to become new focuses, and dies.

Repeat (Maybe):If not all leaves have a dead parent, go back to New Root.

As can be seen above, the algorithm has several subphases, as the focus moves down towards the leaves, and each of these subphases has several sub-sub-phases: Survey, New root, Tournament, Lookup, New Awakening, Refocus, and Repeat (maybe). Internal nodes of the tree have the status dead, focus or live, and leaf nodes either have status awake or asleep. The behavior of each node during each sub-sub-phase will be described.

Survey: Leaves tell parents whether they are active. Intermediate nodes: (live and focus only) Get status from descendants. Remember and (live only) tell parent how many subtrees have one or more active subtrees. Remember which subtree was active if exactly one was.

New root: If a focus has two active subtrees it tells its left (resp. right) child "focus above you= $\langle node \rangle$, you are left (resp. right)". If it has one, tell that one "focus at or below you" and the other "die". It is impossible for a focus to have no active subtree.

Intermediate nodes below a focus (i.e., those nodes that are live) listen to their parents. If one hears "die" it dies. If one hears "focus above = $xxx \ldots$" it relays the message and becomes or remains live. If one hears "focus at or below" it acts as in the paragraph above.

Leaves that receive a "die" message send their parent an "I died" message and prepare to read the next line of the adjacency matrix.

Active leaf nodes record the name of their focus.

Tournament and Lookup: Each leaf contains a mapping M relating the name of each of its ancestors to either nil or the index of a leaf. A sleeping leaf node sends nil to its parent. An awake leaf node i that receives a "focus above you= $\langle node \rangle$, you are left" message sends to its parent either $(\textbf{empty}, i)$ if $M(node) = \textbf{nil}$, or $(\textbf{loaded}, M(node))$. If it receives "focus above you= $\langle node \rangle$, you are right", it sends i to its parent.

A live internal node which receives nil from both children sends the same to its parent; one that receives something else from one child sends that value to its parent, and one that

receives non-nil values from both children sends *either* to its parent. The correctness of the algorithm does not depend on this choice, which can be random, pseudo-random, or consistent.

Each focus receives a message from each child. Say the right child's message is j. If the left child's message is $(\textbf{empty}, i)$, then $(\textbf{record}, focus, i, j)$ is sent to the left child and **nil** is sent to the right. If the left child's message is $(\textbf{loaded}, i)$, then **nil** is sent to the left child and $(\textbf{awaken}, i)$ is sent to the right.

Lookup and New Awakening: Internal nodes relay parents' messages to their children.

If leaf node i receives $(\textbf{record}, focus, i, j)$ it sets $M(focus) \leftarrow j$. If it receives $(\textbf{awaken}, i)$ it awakens. (If i doesn't match, it does nothing.)

Refocus: Each focus sends its children a "become a focus" message and dies. A live node receiving such a message from its parent changes its status to "focus". A leaf receiving such a message form its parents sends the latter an "I died" message.

Repeat (maybe): At all times, a node receiving two "I died" messages sends one upward. If a node receives a "become a focus" message it sends its children a "begin survey" message. Live intermediate nodes relay such a message, and leaf nodes receiving a "begin survey" message proceed as in Survey.

Chapter 6

Use of Additional Techniques – Binary Addition

There are three important classes of circuits for the addition of integers represented as vectors of "bits" in radix 2. These occupy three positions on a spectrum of cost/speed tradeoffs. The fastest and most expensive circuit is a carry-look-ahead adder, described in [Hwa79], which performs addition of two n-bit integers in $\Theta(\log n)$ time using $\Theta(n)$ logic elements. An intermediate circuit, the ripple carry adder, takes linear time and also uses a linear, but smaller, amount of logic. The slowest and cheapest circuit is a serial adder which uses a small constant amount of logic to perform additions in linear time (with a larger proportionality constant than that of a ripple-carry adder).

There are three reasons for studying the synthesis of these addition circuits in TRANSCONS. They are:

- There is a large and interesting space of alternative implementations. If we can not synthesize all of the implementations there is cause to wonder whether TRANSCONS is general enough.
- The three implementations of binary addition to be discussed here fit well into VLSI.
- The synthesis paths shown here demonstrate well how general mathematical knowledge fits together with TRANSCONS techniques to develop VLSI circuits that can not be developed using either alone.

In the remainder of this Chapter we expose the necessary set of synthesis techniques to create implementations of the three solutions to the problem of adding numbers represented as bit vectors.

6.1 Notation

In what follows, we will assume that a problem instance resides in vectors A and B, each containing individual "bits" a_i resp. b_i for $0 \leq i \leq n-1$. The two states of a bit are represented by the values 0 and 1. (This discussion is specialized to binary integers, but any radix can be used by reinterpreting the logical operators in an obvious manner.) We apply logical operators to the values 0 and 1, interpreting 0 as **false** and 1 as **true**. The vectors A (resp. B) represents $\sum_{0 \leq i \leq n-1} a_i 2^i$ (resp. $\ldots b_i 2^i$). The answer is similarly represented in C. We will have occasion to refer to $carry_i$, the carry coming into position i. We use $\bigotimes$ as the symbol for "exclusive OR". We will use n only as the size of the vectors to be added throughout this chapter.

Our starting point for all of the syntheses in this chapter will be the following specification, which produces a vector C given vectors A and B as above:

$$\forall\, 0 \leq i \leq n-1$$
$$c_i = a_i \bigotimes b_i \bigotimes (\exists j < i)[a_j \wedge b_j \wedge (\forall j < k < i)[a_k \vee b_k]]$$

Figure 6.1: The "Standard" Specification of Binary Addition

We will be deriving the following "grade school" specification for binary addition (so called because it corresponds closely to the algorithm taught to grade school pupils for decimal addition) from the standard specification. A derivation of the standard specification from the grade school specification is possible by the methods of [Sto77], but will not be given here.

$$
\begin{aligned}
&carry_0 = 0 \\
&\forall\, 0 \le i \le n-1 \\
&\quad c_i = a_i \bigotimes b_i \bigotimes carry_i \\
&\quad carry_{i+1} = (carry_i \wedge (a_i \vee b_i)) \vee (a_i \wedge b_i)
\end{aligned}
$$

Figure 6.2: "Grade School" Specification for Binary Addition

6.2 Carry Look-ahead Circuit

Consider the standard specification. If we try to use the methods of TRANSCONS that have been described so far to synthesize a carry-look-ahead circuit for addition, we get a circuit with $\Theta(n^2)$ computing elements. The reason for this is the nesting of quantifiers such that the bound variable of the outer quantifier is one end of the range of the inner one. This fact forces the computation of $\Theta(n^2)$ boolean values, namely $(\forall j < k < i)[a_k \vee b_k]$ for each $0 \le j < i \le n-1$ (a total of $n(n-1)/2$ $\langle i, j \rangle$ pairs).

We would like to do better.

6.2.1 Quantifier Levelling

The problem with the standard specification is that it has a pair of nested quantifiers, with the range of the inner quantifier equal to the bound variable of the outer, used as a predicate. Specifically, we have $c_i = a_i \bigotimes b_i \bigotimes (\exists j < i)[a_j \wedge b_j \wedge (\forall j < k < i)[a_k \vee b_k]]$. Evaluating this predicate is expensive. Even reusing values, i.e., using $(\forall j+1 < k < i)[a_k \vee b_k]$ to compute $(\forall j < k < i)[a_k \vee b_k]$, $\Theta(n^2)$ computing elements are required to evaluate the form. However, it is possible to proceed in a series of steps to a form that can be evaluated using only $\Theta(n)$ elements.

First we use the following identity

$$(\forall l < x < u)[P(x)] \equiv \max_{x<u}[\, P(x)] \le l \qquad (\forall\text{-}to\text{-}max)$$

which gives us a tool to express a doubly bounded quantifier as an inequality applied to a max operator. The benefit of doing this is that it reduces the multiplicity of values to be computed. Rather than have a $\Theta(n)$ set of booleans to compute for each j, we have a single $\log n$ bit integer to compute and compare with j.

This doesn't solve the problem. We are left with

$$c_i = a_i \bigotimes b_i \bigotimes (\exists j < i)[a_j \wedge b_j \wedge \underline{\max_{k<i}[\ (a_k \vee b_k)] \leq j}]$$

where the substituted expression is underscored.

e are still faced with the problem of computing the max operator for each of the i's and using a quantifier as a boolean to establish a carry bit for each i. We are therefore going to take advantage of another identity to turn a singly bounded quantifier into a doubly bounded one:

$$(\exists x < u)[P(x) \wedge F(u) \leq x] \equiv (\exists F(u) \leq x < u)[P(x)] \qquad (\textit{constraint-to-binder})$$

providing the following:

$$c_i = a_i \bigotimes b_i \bigotimes \underline{(\exists \max_{k<i}[\ (a_k \vee b_k)] \leq j < i)[a_j \wedge b_j]}$$

where again the new form is underscored.

This does not completely solve the problem, because it would still require a max operation for each i, but it allows us to apply one last identity,

$$(\exists l \leq x < u)[P(x)] \equiv \max_{x<u}[P(x)] \geq l (\exists\text{-to-}max)$$

which gives us

$$c_i = a_i \bigotimes b_i \bigotimes (\max_{j<i}[a_j \wedge b_j] \geq \max_{k<i}[\ (a_k \vee b_k)])$$

Now we have an inequality involving the fruits of two max operations. While each of these max'es must be computed for each i, this is a form that can be treated by the methods of Section 5.2. It is therefore only necessary to build two tree structures in which the computing elements contain $\Theta(\log n)$ logic gates.

The specification is now

$$\begin{aligned} &\forall\, 0 \ \leq i \leq n-1 \\ &\quad c_i = a_i \bigotimes b_i \bigotimes (\max_{j<i}[a_j \wedge b_j] \geq \max_{k<i}[\sim (a_k \vee b_k)]) \end{aligned}$$

It is possible to express this as an inequality between corresponding elements of the results of two parallel prefix computations as follows:

$$\begin{aligned} &\forall\, 0 \leq i \leq n-1 \\ &\quad and_i = a_i \wedge b_i \\ &\quad nor_i = \sim (a_i \vee b_i) \\ &\quad max1and_i = \text{ if } and_i \text{ then } i \text{ else } -\infty \\ &\quad max1nor_i = \text{ if } nor_i \text{ then } i \text{ else } -\infty \\ &\quad maxand_i = \max_{0 \leq j < i}[max1and_j] \qquad * \\ &\quad maxnor_i = \max_{0 \leq j < i}[max1nor_j] \qquad * \\ &\quad C_i = a_i \bigotimes b_i \bigotimes (maxand_i \geq maxnor_i) \end{aligned}$$

There are two parallel prefix trees in the addition parallel structure: one for the variable named *maxand* and another for *maxnor*. The overall structure is shown below.

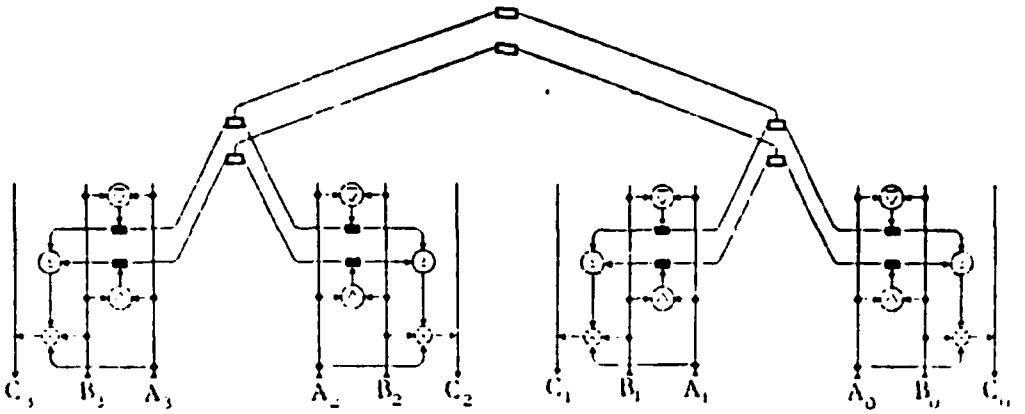


Figure 6.3: Synthesized Look-Ahead Circuit for Binary Addition

There are two important differences between this structure and the standard one of [Hwa79].

- Because of the nature of the parallel prefix network synthesized by TRANSCONS, each node is partially responsible for the choreography in its local region. The importance of this fact is that either the nodes need be big enough to participate in an asynchronous data transfer protocol with a handshake, or a global clock must be provided.

This is not a serious problem because the methods of Section 5.3 enable us to remove dependence on local handshaking, reducing the computation nodes to combinatorial logic.

- Because the parallel prefix trees are required to handle integers in the interval $[0, n]$, the size of the nodes and the width of the data paths within the trees are $\Theta(\log(n))$. In the standard network it would be $\Theta(1)$.

This disadvantage can be alleviated by some careful reasoning, to be described below.

6.2.2 Data Path Width Reduction

To reduce the width of the data paths and still use a parallel prefix network, an associative operation with constant range and domain must be used. We see that this might be possible because either $\max_{0\le j\le i}[max1and_j] = \max_{0\le j\le i+1}[max1and_j]$ or $\max_{0\le j\le i+1}[max1and_j] = i+1$, and similarly for $max1nor$. We have four cases from the four possible values of a_{i+1} and b_{i+1}, or similarly from and_{i+1} and nor_{i+1}. For brevity we will give the name P_i to $maxand_i \ge maxnor_i$. We can show that P_{i+1} depends only on P_i, and_{i+1} and nor_{i+1} from the fact that if neither and_{i+1} nor nor_{i+1} we have $P_{i+1} = P_i$, if and_{i+1} we have P_{i+1}, and if nor_{i+1} we have P_{i+1}. This information can be summarized in the following table:

$maxand_i \ge maxnor_i \Rightarrow$ $\Downarrow$ and_{i+1} nor_{i+1} $\Downarrow$	true	false
and	true	true
nor	false	false
$both^*$	true	true
neither	true	false

(*this is impossible but knowledge of this fact is unnecessary for the argument)

The effect of and_{i+1}, nor_{i+1}, and_{i+2} and nor_{i+2} on the truth of $maxand_{i+2} \ge maxnor_{i+2}$ given $maxand_i \ge maxnor_i$ can also be summarized as below. (Here the impossible combinations have been omitted for brevity.)

$maxand_i \ge maxnor_i \Rightarrow$ $\Downarrow$ and_{i+1}, nor_{i+1}, and_{i+2}, nor_{i+2} $\Downarrow$	true	false
none	true	false
and_{i+1}	true	true
nor_{i+1}	false	false
and_{i+2}	true	true
and_{i+2}, and_{i+1}	true	true
and_{i+2}, nor_{i+1}	true	true
nor_{i+2}	false	false
nor_{i+2}, and_{i+1}	false	false
nor_{i+2}, nor_{i+1}	false	false

A similar (although lengthy) table can be made encompassing three *and*/*nor* value pairs.

We see from the two pair table that any two input bit pairs is an operator that can do one of three things: it can act like an $a_i = b_i =$ **true** bit pair (called $\langle and \rangle$ below), like an $a_i = b_i =$ **false** bit pair (called $\langle nor \rangle$), or like an $a_i \neq b_i$ bit pair (called $\langle other \rangle$). From the above table it can be seen that the binary operator $\oplus = \lambda_{x,y}[\text{if } y = \langle and \rangle \text{ then } \langle and \rangle \text{ elseif } y = \langle nor \rangle \text{ then } \langle nor \rangle \text{ else } x]$ describes the result of combining two adjacent columns. A case analysis on all possible triples shows that this operator is associative. Using $F(a_i, b_i)$ as an abbreviation for if $a_i \wedge b_i$ then *and* elseif $(a_i \vee b_i)$ then *nor* else *other*, it is therefore possible to write $c_i \leftarrow a_i \bigotimes b_i \bigotimes (\oplus_{j<i}[F(a_j, b_j)] = \langle()and\rangle)$. The identity of this operator is $\langle other \rangle$, and $maxand_i > maxnor_i = (\oplus_{0 \leq j \leq i} = \langle and \rangle)$. This is precisely what was needed to perform a parallel prefix summation with constant-width data paths: an associative operator with finite range and domain.

Use of a specification based on this operator will yield a network similar to that of Figure 6.3, except that there will only be a single parallel prefix tree, each bit's carry will be used directly rather than computed from the two parallel prefix trees, and (of course) the widths of the data paths and the size of the nodes will be smaller.

6.3 Ripple-carry and Bit Serial Circuits

Consider our "standard specification" of Figure 6.1. If we apply the quantifier levelling of subsection 6.2.1, we get:

$$\begin{array}{l} \forall\, 0 \leq i \leq n-1 \\ \quad c_i = a_i \bigotimes b_i \bigotimes (\max\limits_{j<i}[a_j \wedge b_j] \geq \max\limits_{k<i}[\sim (a_k \vee b_k)]) \end{array}$$

If we change this to

$carry_{-1} \leftarrow$ **false**
$\forall\, 0 \le i \le n-1$
  $carry_i \leftarrow \max_{j<i}[a_j \wedge b_j] \ge \max_{k<i}[\sim (a_k \vee b_k)]$
$\forall\, 0 \le i \le n-1$
  $c_i = a_i \bigotimes b_i \bigotimes carry_{i-i}$

we can repeat the reasoning of the first part of the last section, obtaining the recurrence relation. This gives us

$carry_{-1} \leftarrow$ **false**
$\forall\, 0 \le i \le n-1$
  $carry_i \leftarrow (a_i \wedge b_i) \vee (carry_{i-1} \wedge (a_i \vee b_i))$
$\forall\, 0 \le i \le n-1$
  $c_i = a_i \bigotimes b_i \bigotimes carry_{i-1}$

This is logically identical to the gradeschool specification of Figure 6.2. More importantly, it can be transformed into

. . .

PC **istype PROCESSORS** $(i), 0 \le i \le n-i$
  HAS c_i
  HEARS *PA* (**USES** a_i)
  HEARS *PB* (**USES** b_i)
  HEARS Pcy_{i-1} (**USES** $carry_{i-1}$)
  TALKS . . .
Pcy **istype PROCESSORS** $(i), -1 \le i \le n-1$
  HAS $carry_i$
  if $i \ge 0$ **then HEARS** *PA* (**USES** a_i)
  if $i \ge 0$ **then HEARS** *PB* (**USES** b_i)
  if $i \ge 0$ **then HEARS** Pcy_{i-1} (**USES** $carry_{i-1}$)
  if $i < n-1$ **then TALKS** Pcy_{i+1} (**SENDS** $carry_i$)

. . .

(**in** Pcy_1) :
  $carry_{-1} \leftarrow$ **false**
(**in** Pcy_i, $0 \le i \le n-1$) :
  $carry_i \leftarrow (a_i \wedge b_i) \vee \ (carry_{i-1} \wedge (a_i \vee b_i))$
(**in** PC_i, $0 \le i \le n-1$) :
  $c_i \leftarrow a_i \bigotimes b_i \bigotimes carry_i$

using the methods of crystalline synthesis described earlier in this thesis. A diagram of the resulting parallel structure (after explicating all values internal to the computations) is shown below.

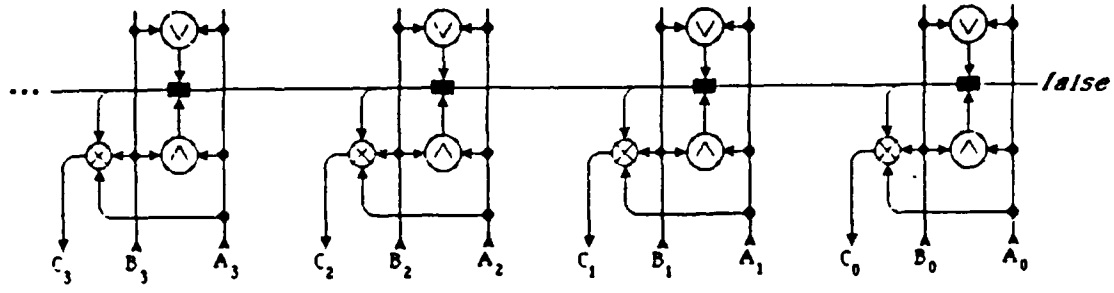


Figure 6.4: Ripple Carry Parallel Structure

The aggregation of Section 3.3 is then applicable. This technique replaces a related series of processing elements by a single element that receives a series of related data. The circuit of Figure 6.4 is an indexed series of identical modules, and identifying corresponding nodes of the series of modules gives the bit serial addition circuit shown below.

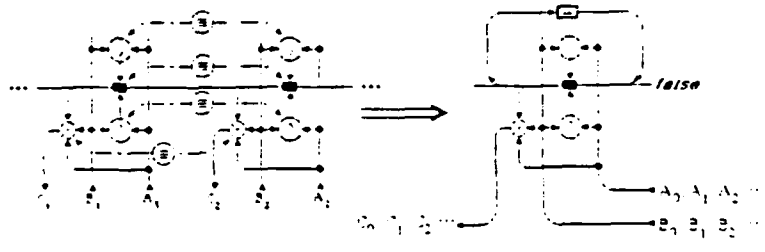


Figure 6.5: Serial Adder

We have seen that quantifier levelling, recurrence relation analysis, crystalline synthesis and aggregation together are sufficient to synthesize the three types of binary addition circuit in common use. It remains to prove the identities used in quantifier levelling. We do this in Appendix Section C.

We have seen that the use of general mathematical identities considerably increases the power of TRANSCONS to synthesize VLSI circuit topologies.

Chapter 7

Conclusions and Summary

7.1 Overview

TRANSCONS is a collection of tools and methods for creating parallel structures from first order logic specifications. This collection of capabilities interacts in various ways to act as a VLSI assistant. We have described the three major divisions of these abilities in the body of the thesis and will summarize them below.

We have also described, and will summarize, the theoretical framework in which a VLSI synthesis system must work. This includes models underlying the various structures that can be created, the theory needed to justify the application of some of the transformational rules of TRANSCONS, and the underlying assumtions that justify the incorporated heuristics.

In this chapter we summarize this thesis by stepping back and describing its results, its main points, suggested future lines of research, and the meaning.

7.2 Essential Points

We claim that we have devised theory for the following forms of transformational synthesis of concurrent structures:

- creation of lattice-like arrays of processors from specifications in a very high level language resembling first order logic;
- modifications of intermediate forms synthesized while creating lattice-like arrays by several techniques called aggregation, virtualization, communication reduction, and chain creation to improve the lattices that eventually result from some that it would be impractical to build to others that would be practical;
- creation of tree structures of processors from FOL specifications using a variant of D&C in which only certain ways of combining problems' partial solutions are permitted;
- use of closures, or communicable functional objects, to facilitate reasoning about bidirectional communication. Instead of reasoning about a message from A to B, reason about a closure sent from B to A. This enables a later transmission from A to B whose effect has already been studied by the closure synthesis process;
- combinations of any of the above, using user supplied aggregations.

We further claim that this is done in a manner so as to facilitate use of expected future advances in theorem proving technology.

We further claim that, using these techniques, a system can be built to do all of the following on a practical basis:

- synthesize regular interconnections of regular arrays of processors, together with programs for the processors, to perform interesting and useful computations which are specified in a very high level language similar to first order logic;

- synthesize systolic arrays;
- synthesize tree interconnections of processors;
- synthesize lower level specifications from any of the above, in some cases down to something suitable for direct VLSI implementation after a placement and routing algorithm (not discussed here) has been run.

TRANSCONS will be an effective tool for allowing integrated circuit designers to cope with the million gates that will shortly be available on a single chip in a manner that does not merely provide larger versions of circuits already available. Automatic programming tools have the ability to bring problem decomposition and program combination knowledge to bear on the task of synthesizing programs too large and complex to write by hand. In an analogous manner, TRANSCONS's ability to bring processor interconnection and problem decomposition knowledge to bear on the task of synthesizing VLSI will make possible the creation of accurate integrated circuits too complex to create by hand.

7.3 Foundations

There are several fundamental points upon which this work rests.

In order to rationally discuss the synthesis of concurrent systems, we must have at least one computation model in mind. TRANSCONS has a series of four models, each more restrictive, more descriptive, and able to use simpler processing elements than the previous one.

There are transformation rules dealing with processor assignment. We initially make very simple assumptions of how things will be organized in a parallel structure, and refine these with other transformation rules.

The models, processor assignments, and refinements are three fundamental parts of TRANSCONS that we have described in previous chapters and will summarize below.

7.3.1 Models

We distinguish four types of processing elements for which it might be desirable to write specifications. These models of processing elements occupy positions along a continuous spectrum of minimum complexity to implement. They range from models that require a general purpose processor capable of rapid "context switching" among several loosely related jobs and having sufficient memory to keep track of each of the jobs and associated data, down to one whose minimum processing element would be a latch connected to a minor piece of combinatorial logic such as an AND gate.

The first part of TRANSCONS transforms a supplied specification from its FOL form into a parallel structure that meets the requirement of the first model only. Following this, a series of transformations can proceed from model to model until the lowest level in which the specification can be solved is reached.

We provide the multiple models because it is not always desirable or even possible to transform a specification to the lowest level. Each level corresponds to an implementation decision that can be correct under certain assumptions of component availability, future transformation of the output by other systems, and finality of the design.

7.3.2 Processor Assignment

The fundamental unit of responsibility in parallel structures that TRANSCONS synthesizes is the array element. If two processors cooperate in any way to produce part of an answer, that cooperation must be expressed as use of values computed in one processor by another. Such uses of communicated values constitute the fundamental unit of computation.

When the specification is delivered to TRANSCONS, it may not be possible to isolate a natural array around which processors can be multiplied. In such a case, whenever there is enough work to make it profitable to split the problem among a large number of processors, there must be a reduction operation within the specification that generates a series of intermediate values. The technique of virtualization, or explicating the multiple assignments to this variable into assignments to different elements of a corresponding array, can be applied.

If the result of applying TRANSCONS to a specification produces a parallel structure that has more processors than desirable, the technique of aggregation can be applied to group together processors that perform similar operations on different data at different times, and arrange for all of the computations in each group to take place in a single processor. There is an aggregation that reverses any virtualization, but when one aggregation can be performed several can be performed (on any but a one-dimensional structure). We have found the technique of performing a virtualization, and then an aggregation that merges diagonal groups of processors (picturing the array of processors geometrically) to be an effective technique for producing one- and two-dimensional systolic arrays.

7.3.3 Connectivity Restructuring

After responsibility for computing array elements is allocated among processors, there is an immediately suggested network of interconnections among the processors, i.e., the particular network that has a wire running directly from the processor computing each datum to each of those processors needing the datum. If this structure is satisfactory, the synthesis process is complete, but in none of the problems we examined has this obvious network been satisfactory.

Two solutions to this problem are reduction of snowballing induced sets, and chain formation.

Snowball reduction involved the discovery that there is a series of processors, each of which requires (in part) some of the set of values required by its predecessor, if any, plus that computed by that predecessor itself. If this and several other minor conditions are met, the set of connections in which each processor is connected to all previous processors in the series can be replaced by the set of connections in which each processor is connected only to its predecessor.

When a large number of processors each need to be connected to an I/O processor, we have an opportunity for chain formation. If it can be established that the set of values required from this I/O processor falls into groups such that each value in each of the groups is required by a distinguished group of processors, then it is possible to first form those groups of processors into a "bucket brigade" chain and then to introduce the values required by the processors in the chain at one end only.

Another opportunity for chain formation arises when the acceptable time for delivering the distributed result of a computation to the I/O processor is sufficiently large to allow the processors to form a bucket brigade to deliver these results rather than having each do it with its own connection. In this case the collection of processors into groups is arbitrary.

7.3.4 Divide & Conquer, and Closures

Many specifications are amenable to solution by parallel structures in which the processors are connected in a binary tree. We use D&C to find tree solutions where they exist. The temptation to do this, of course, is the commonality of form between the nodes of a call graph that would result from execution of a program that fit the D&C scheme and the tree structure.

We have isolated several problems that frequently arise when we try to perform such a synthesis in an obvious manner. In some cases a simple D&C solution would require solution of half of the problem before solution to the other half can be attempted. In other related cases each half of the divided problem can be solved without reference to the other but combination of the two halves' solutions requires significant work. In yet other cases the combination requires the handling of significant data in processors close to the root. All of these problems can be met by changing our view of the problem. For some specifications several D&C solutions are available, but each has one of these three problems.

Instead of considering our task to be computing some array from some other array, we consider our task to be the computation of a functional object which, when applied to a given argument list, produces the desired effect. In the many cases we explored, this solves the group of problems described in the previous paragraph. In addition, this solves another problem facing the use of D&C to synthesize tree structures, i.e., the fact that real problems often require communication both up and down the tree but D&C seems suited to reasoning about upward communication only. It solves this problem by allowing the upward flow of a closure to stand for the downward flow of information.

It is possible to implement a structure including closures using the highest level model of TRANSCONS. We do not wish to leave closures in the finished product, however, because closures' implementation requires extra communication and also requires the structure to

be implemented in this highest level model. We therefore provide methods for removing the closures, modifying a parallel structure that contains them into one that contains explicit communication, replacing the information that flows from the closures' recipients to the hosts when they are applied.

The closure can be used to make D&C a more powerful tool for program synthesis, as well as for making it a practical tool for concurrency synthesis. A frequent difficulty in using D&C, even for sequential program synthesis, is the synthesis of the combine operator, i.e., the program to handle the boundary conditions when combining partial solutions. If we augment the specification to require that the solution to each subproblem supply both the desired information and a closure that performs the gluing operation at the boundaries, the synthesis task becomes easier.

7.3.5 Miscellaneous Techniques

If TRANSCONS is limited to a few specific techniques it will not be a very useful tool. It will synthesize a limited set of parallel structures from their specifications, but it will not extend well because it will not enjoy the use of general mathematical knowledge.

We show, however, that general mathematical knowledge can work well with the TRANSCONS concurrency knowledge to produce better parallel structures than would otherwise be achievable. In one case, the synthesis of networks for solving a version of the Connected Components problem, use of set theory axioms and axioms describing the transitivity of the "same connected component" relation is vital. In another case we explore, the synthesis of circuits for the addition of binary integers, we show purely algebraic techniques without which n bit integers required $O(n^2)$ computing elements to solve. The use of a series of techniques allows this to be successively reduced to $\Theta(n \log n)$ and then

to $\Theta(n)$. Further techniques allow the direct synthesis of a slower network with $\Theta(n)$ elements, and the use of aggregation allows this to be reduced to a circuit with a constant amount of logic.

7.4 Future Work

There is a lot of effort required before this work can be considered complete. We will describe this work from the most direct extensions of work already performed to that portion most requiring future theoretical findings. The latter part will be described in more detail under separate headings.

First, some of the basic knowledge of TRANSCONS has to be codified. Recent improvements to the CHI system to improve the efficiency of knowledge storage and retrieval, and facilitating inclusion of theorem provers, impel modifications to the crystalline synthesis section. The tree rules and data structures exist only in the most rudimentary form because of the newness of the conception of the use of closures for this purpose.

This will yield a TRANSCONS in which the designer has a savant assistant at his disposal, but (s)he must supply assertions and inventions at critical places. The inclusion of backwards inference, or the inference of the form of a functon from assertions about its behavior, will make especially the tree synthesis portion of TRANSCONS much more capable of performing on its own.

There are also four more important extensions to the framework that require some fundamental work. When this is done, we claim that the new rules will mesh well with the prototype system.

7.4.1 Routing Problems

Suppose we need an implementation of the APL statement $A[I] = B$, where all are vectors and where it is known that no two values of I are the same but all are within range. In short, suppose we want to perform a general permutation where each processor knows where to send its value. (The similar case, $A = B[I]$, in which each processor knows where it wants to get its value *from*, can be handled by asserting that each processor knows where to send a closure *to* and having the closure accept a value.) This problem is closely related to sorting.

There are several parallel structures for the permutation/sorting problem. Particularly well-known structures include the flashsort [ReV82], Batcher's Sort [Bat68], and an interesting relatively new one [AKS83]. These are all couched in the language of sorting. There is also a Benes network available; it takes longer to compute Benes network settings than it takes for the other networks to work, but once the settings are computed the permutation is faster [NaS82]. The settings can be represented by a word slightly smaller than twice the size of a processor ID (and they can replace the latter, which can be computed from the former). This can be used when the same permutation will be used repeatedly.

We have a situation that the data flow information would seem to require a complete interconnection among the processors in the case where each of the processors holds one of the values to be permuted and is expected to hold one possibly distinct value after the permutation. It requires deeper knowledge than flow analysis to derive a net such as the shuffle interconnection that is less than complete, but that can accomplish a permutation rapidly. Codification of what it is that a person does when he proves that a smaller network can be adequate for the job, is a good subject for future investigation.

7.4.2 Average- vs. Worst-case Behavior

To develop a good practical solution to the permutation/sorting problem may require ability to reason about average- vs. worst-case behavior. For example, there is an $O(\log n)$ time, $O(n)$ processor sorting algorithm with very reasonable constant factors that has only one problem – it fails with small and asymptotically decreasing probability [ReV82]. Another solution with the same asymptotic behavior and which never fails is described in [AKS83], but the constant factors are comparable to Avogadro's constant, making an implementation obviously impractical.

Our synthesis system should reject the latter solution in favor of the former in cases where $O(\log n)$ time and $O(n)$ processor count is required. We must solve questions of what heuristics to use to generate such parallel structures in cases where sufficiently good worst-case behavior can not be achieved, and when to accept the disparity and report the derived circuit as meeting the designers' requirements.

7.4.3 Efficiency Estimation for Parallel Structures

A third problem is extension and integration of an efficiency expert. It has been shown [Kan79] that sequential programs that arise in practice can be analyzed and an estimate of performance derived. This is especially true where the program is synthesized from higher level specifications and the synthesis process is designed with the needs of the efficiency expert in mind. It is not known whether the same is true for concurrent specifications, with interactions among processors making timing analysis more difficult and with more performance criteria than sequential programs, but it seems likely.

7.5 Accomplishments

The main tangible accomplishment of this work is the beginning of TRANSCONS, a VLSI design assistant. Within the system's limitations, a designer can write input/output specifications in first order logic, guide TRANSCONS at some critical places, and produce a top level block diagram of a circuit that will have that input/output behavior. The blocks it uses are, themselves, specifications that can be exposed to the same process.

TRANSCONS's has two domains of applicability: The first is the synthesis of crystalline parallel structures that are, loosely speaking, those in which the various functional blocks or processors bear fixed relationships to their neighbors. The second is the synthesis of tree structures.

In addition to providing design functions, TRANSCONS provides a framework upon which a structure of further analysis tools can rest. We demonstrate this by exhibiting a synthesis that depends on some forms from set theory, and another that depends on certain theorems concerning quantifiers. These can either be entered by a human as axioms, entered as hypotheses to be proven by an internal theorem prover, or found by the system using the "weakest precondition" work of [Smi83b]. TRANSCONS is designed to grow by accepting new transformation rules, new heuristics as to what rules to select for application, or new theorem proving technology.

Appendix

A TRANSCONS Usage Examples

A.1 Usage Conventions for V

To enter a V program and start working on it, one starts up the CHI system in the manner prescribed for your machine. Currently only INTERLISP/TOPS-20 and SYMBOLICS 36xx implementations are available.

One can then give directives to the CHI system. Such directives are in the syntax of LISP, except that in the function name position there will usually be the locution "#>" which has two purposes: it commands the underlying LISP system to interpret the input stream according to V syntax rules, and it requests CHI to parse the V program into an abstract syntax tree.

In what follows the numbers followed by a dot are prompts; they are typed out by the computer.

1. (#> ⟨the V program⟩)

V analyzes the program and either accepts it, in which case the whole program becomes the *current node*, or rejects it, giving an error message in which the location of the error is highlighted. As with most compilers, the actual error can be in a different place from where the compiler thinks it is. Unlike with other compilers, however, it is possible to get more information from the system concerning the location of the error. Simply type ⟨resume⟩ (LISP machine implementation) or (RETURN) (INTERLISP implementation) to be shown another error guess.

The basic operation on nodes is *rule application*. Other operations include focus changing (to narrow the focus of attention from a whole program to one of its parts or *vice versa*) and various methods of printing out nodes.

In addition to the current nodes there are *named nodes*. It is possible to make a named node by suppling

2. (# > RULE programname ⟨the rest of the program⟩)

After doing this, the name **programname** is attached to the node created by the reader. A node must be named to not disappear when you make a new current node.

There are several commands for manipulating the current node or named nodes, or for printing them out in various formats. The HELP function of the CHI system will help you find them, but we summarize some of the more important ones below:

(PU F) Prints the current node Using the F format (as a list of properties). Not usually necessary except for TRANSCONS implementors.

(PU I) prints the node as a V expression (in the same format as it could be read in). I stands for Infix notation.

(MCN L) interprets a *locator* L and makes it the Current Node. L can either be the name of a named node or a prototype, a number, or an index into the current node. Such an index is either a positive integer if the current node is a list, a property name if the current node is a general object, or 0 if the current node has a parent.

If the locator is the name of a named node or a prototype, that object is made the current node. If the current node is a list, than a locator of i makes the i^{th} element of that node the current node. If it is a general node having a property p than a locator of p will make the current node

the value of the *p* property. Finally, a locator of 0 will select the parent of the current node as the current node.

It is quite possible for any or all of the last three types of selector to be applicable.

PU can be used with a second argument. If this is done than the object found by the locator is printed, not the current node.

The basic operation for making a named node is (DEFV ⟨name⟩ ⟨type⟩ #V ...). This text should appear in a ZWEI buffer or a file and it should be EVALuated or LOADed, respectively. Procedures for doing this are found in the LISP Machine Manual. Valid ⟨type⟩s include RULEs, PROGRAMs, OPERATIONs and PROPERTYs. OPERATIONs and PROPERTYs are primarily the concern of implementers.

RULEs are used to define input/output specifications for TRANSCONS (or any other system implemented in CHI). The TRANSCONS system consists of CHI together with other files consisting of rules (and some properties and operations).

A rule always has a name. Whenever a node to which a rule named *name* should be applied is the current node, incant (AR *name*) to apply the rule. The result of the rule application will be displayed. (If the attempt at application was incorrect, and the rule didn't apply to that node, the system prints "rule *name* did not apply", instead.)

A.2 Specific Rules for TRANSCONS

The first step in transforming a specification to a parallel structure via TRANSCONS is to express the specification in V and enter it into the system. The process of entering the specification in V is briefly discussed here and discussed more fully in [Gre81], [BKPW84].

TRANSCONS operates on specifications whose primary data structures are arrays. In general a higher level data structure would be selected for expressing the task, so the higher level structure has to be transformed. The Chi system, in which TRANSCONS is embedded, has facilities for performing this transformation. See [Kot84].

In the following discussion we will use the matrix multiplication problem to show the various techniques and intermediate states. The initial specification is:

$\forall_i , i \in \{1, \ldots, n\}$
  $\forall_j , j \in \{1, \ldots, n\}$
    $C_{ij} \leftarrow \sum_{k, k \in \{1, \ldots, n\}} A_{ik} B_{kj}$

To describe the bounds of arrays and of the various structures they are transformed into, TRANSCONS has an enumeration object that is slightly less general than that of V to allow for easy handling by a theorem prover. This is described below.

A.2.1 Multiple Objects

The basic stuff of TRANSCONS's parallel structures is the multiple object. Examples of these include array elements, processors, and clumps of processors referred to by **HEARS** clauses.

We use bound variable lists and enumerators to control these multiple objects. We do not supply complete generality, because this would lead to certain problems in proving theorems about the domains, ranges, intersections, unions and subset properties of the sets enumerated by these constructs. An example of an inadmissible form is

A **istype ARRAY** $(b \times c)$, $b \in S,\ c \in T$

The multiplication is inadmissible because admitting it would create problems where questions about induced sets would be undecidable. We limit the sets to those definable in Presburger Arithmetic without multiplication, although we allow the bound variable of an outer quantifier to be considered a constant in inner quantifiers. This turns out to be slightly too restrictive because it is impossible to specify a tree finitely in this language, so we also include special constructs designed solely to specify trees.

Valid bound-variable-list/enumerator pairs for object XXX are of the form $\mathrm{XXX}_{\langle expr \rangle^*}, \langle enumr \rangle^*$ where an $\langle expr \rangle$ is linear in all of the bound variables of the enumerations and an $\langle enumeration \rangle$ is of the form $\langle bound\text{-}variable \rangle \in \langle set \rangle$, where a $\langle set \rangle$ is in turn a set definable in Presburger arithmetic. This means that integer subranges and equality modulo a constant are the basic building blocks, and that definite union and intersection are allowed.

A.2.2 Inclusion of ARRAY declarations

It is necessary to decorate the program with ARRAY declarations to allow for a proper synthesis of multiple processors[1]. It is not necessary to declare all arrays; only those which you would like the system to consider expanding into a multiprocessor configuration.

There are two methods for inserting the necessary ARRAY declarations. One is to edit the ascii form of the specification (the one in the file) and have it reread by the CHI reader. An alternative is to insert the ARRAY declaration at the appropriate point of the specification. First the program must be enclosed in a binding block if it is not already in one. Then go to some point in the bound variable set and use the command (: $\langle arraydeclaration \rangle$). The

[1] In the future, this step will be automated, at least where the bounds of arrays can be determined at "compile time" by data flow.

array declaration will be read by the V reader and inserted at the appropriate point. It is conventional to place all of the array declarations at the beginning of the program, but it doesn't matter as long as they are at the appropriate level of block structure, usually the top. We prefer editing the source because doing this documents the session in a permanent form.

There are ordinary ARRAY declarations describing arrays of data that are computed during the course of a problem, and there are I/O ARRAY declarations that describe arrays of data that come from or go to the outside world. I/O arrays should always be declared because there are important efficiency issues that will be ignored if they are not. These issues are principally those of excessive interconnections being necessary between the I/O channels and the outside world. It will never be assumed that an I/O array is ever resident in more than one processor. For the matrix multiplication problem a proper array declaration is:

bind
A **istype INBOUND ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
B **istype INBOUND ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
result **istype OUTBOUND ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
C **istype ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
do
$\forall_i$,$i \in \{1,\ldots,n\}$
  $\forall_j$,$j \in \{1,\ldots,n\}$
    $C_{ij} \leftarrow \sum_{k,k\in\{1,\ldots,n\}} A_{ik}B_{kj}$
    $result_{ij} \leftarrow C_{ij}$

The inbound and outbound arrays are declared separately. There is yet another declaration for the C_{ij} array because otherwise the system would not be willing to allocate separate processors to the elements of the answer. This separate array is called a *shadow array*. Shadow arrays can be created automatically by a rule named MAKE-SHADOW-ARRAYS. Simply type (AR MAKE-SHADOW-ARRAYS) if a shadow array for each I/O array is

desired. In this case (for simplicity of exposition) we choose not to do this; instead we add a single shadow array by hand[1].

A.2.3 Inclusion of PROCESSORS declarations

After you include all of the ARRAY declarations for arrays that the system should expand, you can have the system introduce tentative **PROCESSORS** declarations. An alternative, to be discussed in a later Section, is to perform a virtualization, which must be done before the **PROCESSORS** declarations are introduced. The rules that introduce **PROCESSORS** declarations are called MAKE-IOPSs and MAKE-PSs.

The console session looks like this

A **istype INBOUND ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
INBOUND ARRAY B_{ij} ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
OUTBOUND ARRAY $result_{ij}$,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
C **istype ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
$\forall_i$,$i \in \{1,\ldots,n\}$
  $\forall_j$,$j \in \{1,\ldots,n\}$
    $C_{ij} \leftarrow \sum_{k,k\in\{1,\ldots,n\}} A_{ik}B_{kj}$
    $result_{ij} \leftarrow C_{ij}$

. (AR MAKE-PSs)

INBOUND ARRAY A_{ij} ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
INBOUND ARRAY B_{ij} ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
OUTBOUND ARRAY $result_{ij}$
C **istype ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
$PROC001$ **istype PROCESSORS** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
    HAS C_{ij}
$\forall_i$,$i \in \{1,\ldots,n\}$
  $\forall_j$,$j \in \{1,\ldots,n\}$
    $C_{ij} \leftarrow \sum_{k,k\in\{1,\ldots,n\}} A_{ik}B_{kj}$

[1] To be strictly correct and to allow to use all of the "intelligence" at its disposal, we should probably have assigned separate shadow arrays to A_{ij} and B_{ij}, too. On a problem of this simplicity this would be pedantic. There is a simple correspondence between the final result achieved by after all rule applications when MAKE-SHADOW-ARRAYS is used and the one we will obtain here, but the former is visually much more complex.

$result_{ij} \leftarrow C_{ij}$

. (AR MAKE-IOPSs)

INBOUND ARRAY A_{ij} ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
*PROC*002 **istype PROCESSORS HAS** A_{ij},$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
INBOUND ARRAY B_{ij} ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
*PROC*003 **istype PROCESSORS HAS** B_{ij} ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
OUTBOUND ARRAY $result_{ij}$
*PROC*004 **istype PROCESSORS HAS** $result_{ij}$,$i \in \{1,\ldots,n\}$,$j \in \{1,\ldots,n\}$
C **istype ARRAY** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
*PROC*001 **istype PROCESSORS** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
HAS C_{ij}
$\forall_i$,$i \in \{1,\ldots,n\}$
$\forall_j$,$j \in \{1,\ldots,n\}$
$C_{ij} \leftarrow \sum_{k,k\in\{1,\ldots,n\}} A_{ik}B_{kj}$
$result_{ij} \leftarrow C_{ij}$

There are several things to note: User input (shown in roman) is not case-sensitive, so any combination of upper and lower case can be used. Output is shown in *other fonts*. As present TRANSCONS uses GENSYM's for processor family names rather than forming elegant ones out of the name of the corresponding variable. Only the family *PROC*001 contains more than one processor. The **USES** and **HEARS** clauses have not been filled in yet.

A.2.4 Adding the HEARS/USES Clauses

After this it is necessary to add the **HEARS** clauses and their **USES** subclauses. The rule that does this is called MAKE-USES-HEARS.

The dialog continues...

. (AR MAKE-USES-HEARS)

A **istype INBOUND ARRAY** (ij) $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
$PROC002$ **istype PROCESSORS HAS** A_{ij} $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
B **istype INBOUND ARRAY** (ij) $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
$PROC003$ **istype PROCESSORS HAS** B_{ij} $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
result **istype OUTBOUND ARRAY** (ij)
$PROC004$ **istype PROCESSORS HAS** $result_{ij}$ $,i \in \{1,\ldots,n\}, j \in \{1,\ldots,n\}$
HEARS $PROC001_{ij}$ $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
(USES C_{ij} $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\})$
C **istype ARRAY** (ij) $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
$PROC001$ **istype PROCESSORS** (ij) $,i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
HAS C_{ij}
HEARS $PROC002$ **(USES** A_{ik} $,k \in \{1,\ldots,n\})$
HEARS $PROC003$ **(USES** B_{kj} $,k \in \{1,\ldots,n\})$
$\forall_i$ $,i \in \{1,\ldots,n\}$
$\forall_j$ $,j \in \{1,\ldots,n\}$
$C_{ij} \leftarrow \sum_{k,k\in\{1,\ldots,n\}} A_{ik}B_{kj}$
$result_{ij} \leftarrow C_{ij}$

A.2.5 Clause Reduction: Simple and Complex

The next step is reducing the awesome connections to $PROC004$ (and those to $PROC002$ and $PROC003$ which become evident when the **SENDS** clauses are added). Normally REDUCE-HEARS can be used, but it won't work here.

. (AR REDUCE-HEARS)

Rule failed to apply.

If there were a clause eligible for such treatment, the operator would have used the CHI structure editor to find the clause and add the reducible property to it. He would have done this by using the editor to make the reducible clause the current node, applying the rule MARK-HEARS-CLAUSE-AS-REDUCIBLE to the node, and then making the whole

program the current node again. This was not done – there was no reducible **HEARS** clause.

The solution to this lies in a rule that partitions the induced sets of instantiations of a **HEARS** clause in such a manner that the clause can be replaced by a pair of clauses; one clause to connect each partition to the outside world, and one that builds a chain within the partition to distribute the data. The *telescoping* property [**Kin82**] is necessary to validate the transformation; at present the operator has to give a little help.

. (PU I)

A **istype INBOUND ARRAY** (ij) $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
$PROC002$ **istype PROCESSORS HAS** A_{ij} $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
B **istype INBOUND ARRAY** (ij) $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
$PROC003$ **istype PROCESSORS HAS** B_{ij} $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
result **istype OUTBOUND ARRAY** (ij)
$PROC004$ **istype PROCESSORS HAS** $result_{ij}$ $, i \in \{1,\ldots,n\}, j \in \{1,\ldots,n\}$
HEARS $PROC001_{ij}$ $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
(USES C_{ij} $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\})$
C **istype ARRAY** (ij) $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
$PROC001$ **istype PROCESSORS** (ij) $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
HAS C_{ij}
HEARS $PROC002$ **(USES** A_{ik} $, k \in \{1,\ldots,n\})$
HEARS $PROC003$ **(USES** B_{kj} $, k \in \{1,\ldots,n\})$
$\forall_i$ $, i \in \{1,\ldots,n\}$
$\forall_j$ $, j \in \{1,\ldots,n\}$
$C_{ij} \leftarrow \sum_{k,k\in\{1,\ldots,n\}} A_{ik}B_{kj}$
$result_{ij} \leftarrow C_{ij}$

. STEPS

(... lists the parts of the program

)

. 8

$PROC001$ **istype PROCESSORS** (ij) $, i \in \{1,\ldots,n\},\ j \in \{1,\ldots,n\}$
HAS C_{ij}

HEARS *PROC*002 (**USES** A_{ik} ,$k \in \{1,\ldots,n\}$)
HEARS *PROC*003 (**USES** B_{kj} ,$k \in \{1,\ldots,n\}$)

. HEARS-CLAUSES

(**HEARS** *PROC*002 (**USES** A_{ik} ,$k \in \{1,\ldots,n\}$)
HEARS *PROC*003 (**USES** B_{kj} ,$k \in \{1,\ldots,n\}$))

. 1

HEARS *PROC*002 (**USES** A_{ik} ,$k \in \{1,\ldots,n\}$)

. (AR MARK-HEARS-CLAUSE-FOR-TELESCOPING)

Please give a V expression for the telescopes

(# > PROC002$_{ik}$,k $\in \{1,\ldots,n\}$)

Please give a V expression condition for a processor to be connected to the outside world

(# > PROC002$_{i1}$)

Please give a pair of V expressions describing the connections

(# > PROC002$_{ik}$) **HEARS** (# > PROC002$_{i,k-1}$)

.

The marking process causes no change in the appearance of the program under a (PU I). The new properties are invisible. They would show up under a (PU F) performed when the current node was the marked **HEARS** clause, and of course they are visible to the rules.

Now that properties have been attached to the chosen **HEARS** clause two rules can be applied. One rule, MAKE-CHAIN, produces a new **HEARS** clause building the chains of processors; the other, INPUT-ONLY-AT-BEGINNING-OF-CHAIN, takes advantage of this to reduce the power of the original clause by giving it a condition that makes it

apply only to the processor at the beginning of a chain. This also moves the —USES clause, removes the now-spurious properties, and does a bit more housekeeping. There is an analogous rule, OUTPUT-ONLY-AT-END-OF-CHAIN, that reduces the power of a **HEARS** clause in an output device.

The function of exploiting telescopes to reduce a **HEARS** clause is split into two rules for two reasons: the second rule is different for input and output, and it will not always be necessary to perform both operations. If a chain is already there one of the I/O-ONLY... rules can be applied.

The dialog continues...

. 0

. 0

. 0 to get to the top level (computer's printouts omitted for brevity.)

. (AR MAKE-CHAIN)

*PROC*001 **istype PROCESSORS** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
HAS C_{ij}
if $j > 1$ **then**
HEARS $PROC001_{i,j-1}$
HEARS *PROC*002 (**USES** A_{ik} ,$k \in \{1,\ldots,n\}$)
HEARS *PROC*003 (**USES** B_{kj} ,$k \in \{1,\ldots,n\}$)

. (AR INPUT-ONLY-AT-BEGINNING-OF-CHAIN)

*PROC*001 **istype PROCESSORS** (ij) ,$i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,n\}$
HAS C_{ij}
if $j > 1$ **then**
HEARS $PROC001_{i,j-1}$
if $j > 1 \wedge j < n$ **then**
LINKS $PROC001_{i,j-1}$, $PROC001_{i,j+1}$
(**PASSES** A_{ik} ,$k \in \{1,\ldots,n\}$)
if $j = 1$ **then**

HEARS *PROC*002 (**USES** A_{ik} , $k \in \{1, \ldots, n\}$)
HEARS *PROC*003 (**USES** B_{kj} , $k \in \{1, \ldots, n\}$)

Voila! The size of the induced set of **HEARS** *PROC*002... has been reduced from $O(n)$ to 1.

A.2.6 State of the Implementation

We divide the state of the implementation into three parts; that which is done, that which is just engineering and likely to be done within a few months, and that which may require more serious thought.

Already Done

We have integrated most of the "prototypes" (internal descriptions of processor structures) and rules for lattice synthesis, except for the selection of individual processors' programs, into CHI and have performed test syntheses. The places where a theorem prover is necessary have been replaced by a dialog with the user such as "Is ... a theorem?" and "What must the expression A be to make $P(A)$ a theorem?".

Immediate Next Steps

Next we integrate the prototypes for tree structures, and those rules for same that do not require backwards inference. A theorem prover, probably LMA, will be connected to the (THEOREM ...) calls in the lattice synthesis section. There is one instance of backwards inference in the lattice synthesis section, but this need can probably be met with an *ad hoc* procedure.

We will implement MightyMouse (see below).

And Finally, ...

We integrate backwards inference, which will be able to find a form that has a given property. This is, of course, a difficult problem, but one in which some progress has been made. Research is being done in this because of its importance to divide & conquer. See, for example, [Smi82].

In addition we will supply rules for the rephrasing of array problems as closure computation problems.

These steps together will make the tree synthesis subsystem complete.

A.2.7 MightyMouse

Exploring a fairly large program in Chi in general and TRANSCONS in particular can be a laborious undertaking because of the need to move around the structure. To do this the user has to know what property he wants to use for his descent. This is rather more internal knowledge than we feel that a user should need; he should have a fair idea of what text he wants to think about, but how the levels of the tree divide and what the names of things are should not be part of this knowledge.

MightyMouse is entered by evaluating (mmouse) to edit the current node or (mmouse ⟨*node*⟩) to edit ⟨*node*⟩. When you do this, two panes will be flashed on the screen:

- the Position Pane, which is used for noodling through the data. This pane shows a prettyprinted version of the data. Parts of it are "mouse-sensitive". This means that whenever you position the mouse so that the pointer points to a character, the computer will know what commands apply to. The indicated text is the smallest block of text corresponding to one tree node that includes the character pointed to by the mouse.

When you "click" on indicated text a tree node is chosen as follows: the left button selects the highest tree node represented by the text, the middle button selects an intermediate node, and the right button selects the lowest. If there are more than three choices and the middle button is used, a menu "pops up" that invites the user to select a choice.

Once a node is selected a menu pops up with several options: you can make the node current, select a subnode, select a supernode, change the value of the slot that the node sits in, or choose to apply an applicable rule. Rules can be applied in automatic or semiautomatic mode. In automatic mode, the rule is fully applied, i.e., it is applied at all places where it can be applied. In semiautomatic mode, every time a pattern match is successful the matching node is highlighted and the user can either click one button to apply the rule, another button to skip that application, or a third button to exit completely.

- the History Pane, which shows the last sixteen current nodes. You can make any one of them current. The history pane remembers its history from one call of MMouse to the next, so several unrelated data can be explored together.

B Correctness Considerations

Below we show the formal reasoning required to validate the primary rule that performs **HEARS** clause reduction. The motivation for this rule is the fact that the interconnections inferred from the data flow information can be unacceptably rich. We want to reduce these interconections, hence the name *REDUCE−HEARS* of the rule primarily responsible for this change. We want neither to cut off a processor from information it needs to compute its answer, nor to create such circuitous paths for data that the converted architecture is significantly slower than the original. "Significantly slower" here will mean "slower, by more than a constant factor".

The *REDUCE–HEARS* transformation establishes a pipeline from one processor through a series of other processors instead of a wire from the first processor to each of the other processors. We intend to show that the use of this transformation neither renders the specification incorrect nor less efficient (up to a constant factor). We will use two separate theorems.

The first theorem claims that if a specification is correct, here meaning that all of the data it needs to do its work is available at some time, than the specification resulting from an application of *REDUCE–HEARS* will also be correct in this sense. The rule does not cause other changes to the specification except for the replacement of **HEARS** clauses with smaller **HEARS** clauses in combination with **PASSES** clauses.

The first lemma to the second theorem makes a much broader claim than is necessary merely to limit this rule to a constant factor slowdown. It states that whenever there is a collection of processors such that the longest path length is $l(n)$, the largest number of values computed in one processor is $v(n)$, the highest in-degree of any processor is $i(n)$ and the largest amount of calculation per input value is $c(n)$, the connections form a DAG, and a couple of other reasonable conditions are met, then the runtime of the parallel structure is at most $O(i(n)v(n)l(n)c(n))$. The second theorem will then claim that this establishes a speed-preserving property for *REDUCE–HEARS*

rule *REDUCE–HEARS* (*stmt*) **TRANSFORM**
stmt : '*PNAME* **istype PROCESSORS** (\$*PDV*) \$*PENUMER*...
if *COND*1 **then**
HEARS $PNAME_{\$HBV}$ \$*HENUMER*
(**USES** $UV_{\$UBV}$\$*UEN*,...)...'
∧ (**THEOREM**
$(IS1 = \{HBV : HENUMER[PDV \setminus PDV_1]\}$;(1)
$\wedge\ IS1a = \{HBV : HENUMER[PDV \setminus PDV_2]\}$;(2)
$\wedge\ IS2 = \{PDV : HENUMER \wedge HBV = PDV\}$;(3)
$\wedge\ PROC1 = \{PDV_1\}$;(4)
$\wedge\ PROC2 = \{PDV\}$;(5)
$\wedge\ PROCh = \{HEXPR\}$;(6)

$$\wedge\, PROCh_i = \{HIEXPR\} \quad ;(7)$$
$$\wedge\, (IS1 \bigcap IS1_a) \in \{\emptyset\ IS1\ IS1_a\} \quad ;(8)$$
$$\wedge\, ((\emptyset \subseteq IS1 \subset IS1_a \wedge COND1) \quad ;(9)$$
$$\Rightarrow IS1 \bigcup PROC1 = IS2) \quad ;(10)$$
$$\wedge\, (COND2 \leftrightarrow COND1 \wedge IS1 \bigcup PROCh = IS2) \quad ;(11)$$
$$\wedge\, (\sim COND2 \Rightarrow \nexists PDV_3[IS1 \subset \{HBV : HENUMER[PDV \backslash PDV_3]\}]) \quad ;(12)$$
$$\wedge\, (COND3 \leftrightarrow COND2 \wedge COND1[PDV \backslash HIEXPR]) \quad ;(13)$$
$$\wedge\, (HIEXPR[PDV \backslash HEXPR] = PDV))) \quad ;(14)$$

$\rightarrow$

stmt : '*PNAME* **istype PROCESSORS** (\$*PDV*) \$*PENUMER* ...
  if *COND*3 **then**
    LINKS $PNAME_{\$HEXPR}, PNAME_{\$HIEXPR}$
      (**PASSES** $UV_{\$UBV}\$UEN, \ldots$)
  if *COND*2 **then**
    HEARS $PNAME_{HEXPR}$ ... '

The intent of this rule, especially that of the call of the theorem prover that makes up its bulk, is not obvious. We will explain it before proving theorems about it:

As in all TRANSCONS transformation rules, free variables on the left hand side of the rule (in this case, everything above the "$\rightarrow$") are implicitly existentially quantified. The objects $IS1$ through $PROCh_i$ receive setformer-valued expressions, some of which (i.e., PROCh) form singleton sets. $COND2$ and $COND3$ are instantiated to boolean predicates with free variables chosen from PDV by a similar process. "Backward inference", or the determination of the form of an expression from assertions about its value, must be used in four cases; to get values for $HEXPR$, $HIEXPR$, $COND2$ and $COND3$.

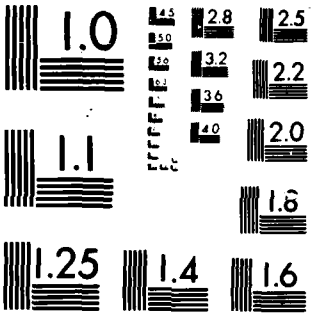
The setformer expressions of the theorem above give us sets of vectors of values, whose dimension is the same as PDV which is the vector of bound variables in the original **PROCESSORS** declaration. The subscripts on instances of PDV in the setformer produced distinguished names, so (for example) if the first element of PDV is i then the first element of PDV_1 is i_1, a different object that need not unify to the same thing i unifies to.

AD-A164 022 KNOWLEDGE-BASED TRANSFORMATIONAL SYNTHESIS OF EFFICIENT 3/3
STRUCTURES FOR CO.. (U) KESTREL INST PALO ALTO CA
R M KING 30 SEP 85 KES. U. 85. 5 AFOSR-TR-85-1259
UNCLASSIFIED F49620-85-C-0015 F/G 9/2 NL

END

FILMED

DTIC



Theorem B.1 *Suppose REDUCE–HEARS applies to a given* **HEARS** *clause. Then every processor of the specification resulting from the application will have all data available to it that the corresponding processor had in the original specification.*

Proof: REDUCE–HEARS applies to **HEARS** clause H_0. The **THEOREM** conjunct contains a large conjunction that implies several things. By lines 1, 2 and 8 we have that two induced sets of H_0 (called $IS1$ and $IS1a$) are either disjoint or telescoping (one contains the other). By 1, 2, 3, 9 and 10 we have that if two induced sets telescope and one is strictly smaller than the other there is a singleton set (called $PROC1$ which we can add to the smaller one to make a different induced set (called $IS2$). Line 4 gives the name PDV_1 to the indices of the processor comprising the singleton set. Line 6 asserts that there is an index expression $HEXPR$ that generates the singleton set $PROCh$ and that has PDV for free variables, such that $PROCh$ indexes the processor that must be added to $IS1$ to get $IS2$, and that this is possible whenever $COND2$ is true. Line 12 asserts that $COND2$ is true whenever such a processor can be found. Lines 7 and 14 assert that the $HEXPR$ mapping has an expression that is its inverse, called $HIEXPR$, and line 14 asserts that a given processor is mapped into by the $HEXPR$ relationship for some other processor. It must be shown that wherever the original specification had a **HEARS** A clause that allowed data to flow from point A to point B and satisfy a **USES** clause, the new specification will have a **HEARS** C clause, and either C = A or there is an unbroken chain of processors that **PASS** the data from A to B. We will do this by induction on the size of the induced set of H_0.

If B **HEARS** one processor then clearly the only way to satisfy lines 9-11 is with that processor as $PROCh$ (and $IS1 = \emptyset$). If B **HEARS** more than one, then it hears $IS2$ and by 9-11 there is a processor $PROCh$ whose induced set is $IS1$ such that $IS1 \uplus PROCh = IS2$, so $PROCh$'s induced set is smaller than $IS2$. $PROCh$ therefore has access to the data it needs.

We further know that $PROCh$ needs information available only from $IS1$ and from all of $IS1$. The passes clause of the third line from the bottom of the rule assures that information available to $PROCh$ will be available to $PROC2$ as well. This completes the induction and the proof. ∎

Now we show that there is not too much of a slowdown assuming certain reasonable restrictions on the computation performed in the processors.

First we need a definition:

Definition B.1 *Each processor performs a computation. The form of the computation can be represented as a tree. Call the highest node(s) with execution time of* $O(1)$ `outermost fast nodes`. *Each outermost fast node is either the highest node of the computation or is used an asymptotically nonconstant number of times. Call the sets of values used by the several uses of outermost fast nodes* `fast sets`.

It is clear that the size of a fast set must be $O(1)$.

Now we can show that the asymptotic performance of the parallel structure that results from $REDUCE-HEARS$ is equal (to a constant factor) to that of the unreduced parallel structure.

Lemma B.2 *Suppose there is a collection of processors such that the longest path length is* $l(n)$, *the largest number of values computed in one processor is* $v(n)$, *the highest in-degree of any processor is* $i(n)$, *the connections form a DAG, and the fast sets are disjoint and contain at most one datum from each input path. If use (by the enclosing node) of the value generated by a fast node takes* $O(1)$ *time, and if a processor that receives at most* v *values on any of its input lines sends at most* v *copies of values that it has received on its output lines, then all processors finish their jobs within* $O(i(n)v^2(n)l(n))$ *units of time. We*

assume that during each time unit a processor can receive one value from each input, send one copy of a previously received value on each output, and do some computation.

Proof: By induction on the length of the longest path ending in a processor. Let $F(P)$ be the length of this path for processor P (the length of a path is the number of *nodes* on the path, so $F(q) = 1$ if q has no inputs). The induction hypotheses are: ($\aleph 1$) that P completes its work in $O(i(n)v^2(n)F(P))$ time, and ($\aleph 2$) that by $F(P) + O(j)$ it has received j values on each of the input lines that has that many values to send. Processor P receives at most $v(n)F(P)$ input values on each input line, and per ($\aleph 2$) it will take at most $F(P) + c_2rk$ time units to receive rk values from an input line that is due to send k.

Suppose $r = \frac{i(n)-s}{i(n)}, 0 < s \leq 1$. After $F(P) + c_2rk$ it will have received rk values, which is all but $\frac{s}{i(n)}k$, and by one time unit later it will have retransmitted them all. This validates ($\aleph 2$) for processors in $\{Q : F(Q) \leq F(P) + 1\}$.

We have a second induction on the amount of time that has passed. Say the constant of ($\aleph 1$) is c_1. $c_1 \geq c_2$ or processor P would be able to complete its work before it received all of its input. If P must complete and use m fast node computations then ($\aleph 3$) is that by the time $c_1v^2(n)F(P)\frac{i(n)-s}{i(n)}$ it will have been able to complete $\lceil sm \rceil$ of them. The base case $s = 0$ requires nothing, and at the time $c_1v^2(n)F(P)\frac{i(n)-s}{i(n)}$ there will be at most $\lceil (1-s)m \rceil$ fast sets for which all values have not been received. If $\aleph 3$ is true for a given s then it must be true for $s + \frac{1}{m}$; the information will be available by $\aleph 2$ and it will be possible to use it sufficiently fast for some c_1 by $\aleph 3$.

$\aleph 1$ is immediate from $\aleph 3$, and the theorem is immediate from $\aleph 1$. ∎

Theorem B.3 *The asymptotic speed of the result of a reduction is equal to that of the unreduced parallel structure.*

Proof: The reduced parallel structure finishes in $O(i(n)v^2(n)l(n))$ units of time. For any single reduced structure $i(n)$ and $v(n)$ will be constant functions, so it must only be shown that the performance of the unreduced structure is no better than $O(l(n))$. But by the snowballing property a chain of length $l(n)$ can only have arisen if there was a processor in the unreduced structure that receives $l(n)$ values, which it would certainly require $O(l(n))$ time to process. ∎

C Quantifier Levelling Proofs

We used three identifiers on quantifiers during the quantifier levelling of Chapter 6. The need for these identies arises from the fact that we have a predicate with a quantifier whose bound variable is bounded on both ends and which occurs in a context in which a series of values obtained by varying *both* bounds is desired. The computation is expensive because a two dimensional array of values is needed, and nothing analogous to a parallel prefix operation is directly available.

The values that are found in this two dimensional array are by no means independent, and with some manipulation the array can be "squashed" into a pair of vectors. What we accomplish by the identities that we exploit in Section 6.2 and demonstrate here is this squashing of the array by summarizing intervals over which predicates with bounded quantifiers are true as pairs of integers.

We display the identities below and then give the proofs.

$$\forall l < x < u[P(x)] \equiv \max_{x<u}[\ P(x)] \leq l \qquad (\forall\text{-}to\text{-}max)$$

$$\exists x < u[P(x) \wedge F(u) \leq x] \equiv \exists F(u) \leq x < u[P(x)] \qquad (constraint\text{-}to\text{-}binder)$$

$$\exists l \leq x < u[P(x)] \equiv \max_{x<u}[P(x)] \geq l \qquad (\exists\text{-}to\text{-}max)$$

Theorem C.1

$$(\forall l < x < u)[P(x)] \equiv \max_{x<u}[\ P(x)] \leq l \qquad (\forall\text{-}to\text{-}max)$$

Proof: By convention, $max_{x<u}[\mathbf{false}] = -\infty$ where $(\forall i)[-\infty < i]$. We have

$$\max_{x<u}[P(x)] = y \equiv P(y) \wedge (\forall y < x < u)[\ P(x)] \qquad (1)$$

and

$$\max_{x<u}[P(x)] < y \equiv (\forall x < u)[\ P(x)] \vee (\exists w < y)[P(w) \wedge (\forall w < x < u)[\ P(w)]] \qquad (2)$$

These are from the definition of $\max_{x<u}[P(x)]$ as that x such that $P(x)$ is indeed true and that $P(x)$ is false for any $x < x < \mathrm{u}$.

We also have

$$w < y < u \Rightarrow ((\forall w < x < u)[\ P(w)] \equiv ((\forall w < x \leq y)[\ P(w)] \wedge (\forall y < w < u)[\ P(w)])) \qquad (3a)$$

and

$$y < u \Rightarrow ((\forall x < u)[\ P(x)] \equiv ((\forall w \leq y)[\ P(w)] \wedge (\forall y < w < u)[\ P(w)])) \qquad (3b)$$

as these forms merely split a quantified predicate, which is a statement about a range of integers, into a conjunction of statements about portions of that range.

So (2) becomes:

$$\begin{aligned}\max_{x<u}[P(x)] < y \equiv \quad & (\forall x \leq y)[\ P(x)] \wedge (\forall y < w < u)[\ P(w)] \\ & \vee (\exists w \leq y)[P(w) \wedge (\forall w < x \leq y)[\ P(x)] \wedge (\forall y < x < u)[\ P(x)]]\end{aligned} \qquad (4)$$

Factoring (4), (and observing that the inner quantifier of (4) is independent of the outer quantifier's bound variables) we get

$$\begin{aligned}\max_{x<u}[P(x)] < y \equiv \quad & (\forall x \leq y)[\ P(x)] \vee (\exists w \leq y)[P(w) \wedge (\forall w < x \leq y)[\ P(x)]] \\ & \wedge (\forall y < w < u)[\ P(w)]\end{aligned} \qquad (5)$$

but the first conjunct of (5) is true by the definition of $\forall$ in terms of $\exists$, the law of the excluded middle, and the fact that any nonempty subset of a finite set of integers has a maximal element. So we have $\max_{x<u}[P(x)] \leq y \equiv \mathbf{true} \wedge (\forall y < w < u)[\ P(w)]$. ∎

The next identity simply restates a singly boundedly quantified predicate which includes a restriction on the bound variable of the quantifier as a doubly bounded quantified predicate without the restriction. This is obvious from the definitions.

Theorem C.2

$$(\exists x < u)[P(x) \wedge F(u) \leq x] \equiv (\exists F(u) \leq x < u)[P(x)] \qquad (\textit{constraint-to-binder})$$

Proof: Immediate from the definition $(\exists l < x < u)[P(x)] \equiv (\exists x)[P(x) \wedge l < x \wedge x < u]$. ∎

Theorem C.3

$$(\exists l < x < u)[P(x)] \equiv \max_{x < u}[P(x)] > l \qquad (\exists\textit{-to-max})$$

Proof: This is the dual of (∀-to-max). ∎

D Theorem Reduction Forms

Below are the rules used to transform TRANSCONS's (**THEOREM** ...) forms into Presburger Arithmetic with restricted quantifier depth:

$possible-theorem$:= $sboolean \mid sboolean\ [\ \wedge / \vee / \Rightarrow\ etc.\ sboolean] \mid \sim sboolean$

$sboolean$:= $set = set \mid bset \in explicit-set-of-sets \mid set \supset set$
$\mid |bset1| + constant = |bset2| \wedge bset2 \supset bset1$

set := $set\ [\bigcup / \bigcap / -\ set] \mid bset$

$bset$:= (ordinary setformers)

The sufficiency of these concts can be argued from the following observations:

- Telescoping and Snowballing can be expressed in this language, provided the underlying sets are expressible with legal setformers. (This proviso will not be reated in what follows.)
- The answers to value-flow questions (eg. REACHES) can be phrased in this form. A node reachable from two places will generate a $\bigcup$, an IF may generate an $\bigcap$ or an $\wedge$ or a $-$, and nested loops generate setformers with multiple bindings.
- A question of whether a processor from a given set is connected to a processor from another given set can be asked in this form if the two sets of processors are expressable by ITCONST rules.

We have a form, (**THEOREM** ...), where the argument is a V expression conforming to the above syntax. Rules, to be displayed below, are used to reduce these expressions to longer but simpler ones from P. A. This language is adequate for the REDUCE-HEARS rule, which reduces the communication paths of an amenable network to a smaller one. We claim it will prove to be adequate for future needs.

The form of a setformer is {⟨expr⟩ : (⟨bvlist⟩)⟨exprlist⟩ | ⟨predicate⟩}.

⟨expr⟩ is an expression linear in all variables of (⟨bvlist⟩). (⟨bvlist⟩) is a list of variables. A (lexical) binding scope is created for each variable that includes the whole set former

(and no more). The ⟨exprlist⟩ had better be a list of expressions. For simplicity we require that each be of the form *bv* ∈ *set* where *bv* is from the ⟨bvlist⟩ and set is itself either an admissible set former, an integer subrange, or the finite union/intersection of the above. "|⟨predicate⟩" is optional, but where present it is a boolean expression. (When not present, "true" is used.)

The (**THEOREM** ...) function receives a general V expression and applies the transformations given above until no change. It passes the result, a P. A. expression, to the real theorem prover.

Below we list the rules for converting forms from the (**THEOREM** ...) language. The transformations below all preserve semantics. If we group the allowable connectives as follows: $|set1| + \langle const \rangle = |set2| \ldots, \bigcup / \bigcap, \in, =, \supset$ we can easily observe that the transformations below always strictly reduce the number of occurences of the highest ranked connective that they touch at all. Since an instance of the highest ranked non-P. A. form in an expression is always reducible, the process terminates with no such forms left.

Each of the rules given below performs some of the reduction by decreasing the number of places it applies without increasing the number of places that rules appearing earlier in this list apply.

rule SPECIFIC-SIZE-DIFFERENCE-SUPERSET (*s*) **TRANSFORM**
$s : `\|\|set1\|\| + \langle constant \rangle = \|\|set2\|\| \wedge set2 \supset set1'$
→
$s : `\exists X_1, X_2, \ldots X_{\langle const \rangle} \forall 1 \leq i, j \leq \langle const \rangle$
$[\ X_i \neq X_j \wedge \forall y \in set2 [y \in set1 \vee Y = X_1 \vee Y = X_2 \vee \ldots \vee Y = X_{\langle const \rangle}]]'$

rule TEST-IF-ANY-EQUAL (*s*) **TRANSFORM**
$s : `x \in \{y_1, y_2, \ldots, y_i\}'$
→
$s : `x = y_1 \vee x = y_2 \vee \ldots \vee x = y_i'$

rule SETFORMER-INCLUSION-TO-FUNCTION-TEST (*s*) **TRANSFORM**
$s : `\{F_1(b_1) : b_1 \in s_1 \mid T_1(b_1)\} \supset \{F_2(b_2) : b_2 \in s_2 \mid T_2(b_2)\}'$
→
$s : `\forall b_1[\ b_1 \in s_1 \wedge T_1(b_1) \rightarrow \exists b_2[b_2 \in s_2 \wedge T_2(b_2) \wedge F_1(b_1) = F_2(b_2)]]'$

rule SET- = -TO-2-WAY-INCLUSION (*s*) **TRANSFORM**
$s : `set1 = set2'$
→
$s : `set1 \supset set2 \wedge set2 \supset set1'$

rule EMPTY-SET-FROM-SETFORMER (*s*) **TRANSFORM**
$s : `\{F(b) : (b)b \in s \mid T(b)\} = \emptyset'$
→
$s : `\forall b[b \sim\in s \vee \sim T(b)]'$

rule UNION-TO-OR (*s*) **TRANSFORM**
$s : `a \in x \bigcup y'$
→
$s : `a \in x \vee a \in y'$

rule INTERSECTION-TO-AND (*s*) **TRANSFORM**
$s : `a \in x \bigcap y'$
→
$s : `a \in x \wedge a \in y'$

rule INTEGER-SUBRANGE-MEMBERSHIP-TO-INEQUALITIES (*s*) **TRANSFORM**
$s : `a \in \{low \ldots high\}'$
→
$s : `a \geq low \wedge a \leq high'$

rule MEMBER-SETFORMER (*s*) **TRANSFORM**
$s : `a \in \{F(b) : b \in s \mid T(b)\}'$
→
$s : `\exists b[b \in s \wedge T(b) \wedge a = F(b)]'$

rule TWO–LAYER–MEMBER–SETFORMER (s) **TRANSFORM**

s : '$a \in \{F(b,c) : b \in s, c \in t(b) \mid P(b,c)\}$'

$\rightarrow$

s : '$\exists b[\ b \in s \wedge \exists c[c \in t(b) \wedge P(b,c) \wedge a = F(b,c)]]$'

References

[AUl72] Aho and Ullman, "The Theory of Parsing, Translation and Compiling"; Volume 1, Prentice-Hall, *pp. 314-320*

[AHU74] Aho, Hopcroft and Ullman, "The Design and Analysis of Computer Algorithms", Addison Wesley*pp. 67-68*

[AKS83] M. Ajtai, J. Komlós and E. Szemerédi, "An $O(n \log n)$ Sorting Network" *Proceedings of the 15th ACM Symposium on Theory of Computing, pp. 1-9*, 1983

[AG79] W. Armstrong J. Gecsei, "Architecture of a Tree-Based Image Processor" *Tech Report, University of Montreal, Publication 291*, 1979

[AHe77] R. Atkinson and C. Hewett, "Synchronization in Actor Systems" *Symposium on Programming Languages*, Jan 1977

[Bar82] C. Bartet, "Policy-Protocol Interaction in Composite Processes" *MIT AI Lab Memo 692*, September 1982

[Bat68] K. Batcher, "Sorting Networks and their Applications" *AFIPS Spring Joint Computer Conference, pp. 307-314*, 1968

[BKu79] J. Bentley and H. Kung, "Two Papers on a Tree-Structured Parallel Computer" *Carnage-Mellun University Tech Report CMU-CS-79-142*, September 1979

[BLe82] Sandeep N. Bhatt and Charles E. Leiserson, "How to Assemble Tree Machines" *Proceedings of the 14th Symposium on Theory of Computing, pp. 77-83*, 1982

[Bro82] Thomas C. Brown, "Inference Requirements Analysis and Implementation Proposal for Two Synthesis Rules", Kestrel Tech Report #KES.U.82.10, *1982*, Chapter 2

[Brw80] Sally A. Browning, "The Tree Machine: A Highly Concurrent Computing Environment", *California Institute of Technology Ph. D. Thesis*, 1980

[BSdJ82] R. Byrd, S. Smith and S. de Jong, "An Actor-Based Programming System" *IBM Research Report #RC 9204 (#40424)*, January 1982

[CMi78] K. Chandy and J. Misra, "Specification, Synthesis, Verification and Performance Analysis of Distributed Programs; a Case Study; Distributed Simulation" *University of Texas, Austin Tech Report TR-86*, November 1978

[CMe82] M. Chen and C. Mead, "Formal Specifications of Concurrent Systems" *Technical report 5042:TR:82, California Institute of Technology*, 1982

[Cho-82] Y. Choo, "Hierarchial Nets – A Structured Petri Network Approach to Concurrency" *Cal Tech Report TR:5044:82*, November 1982

[Chu51] A. Church, "The Calculi of Lambda-Conversion" *Annals of Mathematical Studies # 6*, Princeton University Press

[Cla78] E. Clarke, "Concurrent Programs are Easier to Verify than Sequential Programs" *Duke University Tech Report CS-1978-6*, July 1978

[Cli81] W. Clinger, "Foundations of Actor Semantics", PhD Thesis, *MIT AI Lab Tech Report AI-TR-633*, May 1981

[CLW79] K. Chung, F. Luccio and C. Wong, "A Tree Storage Scheme for Magnetic Bubble Memories" *IBM Research Report # RC 8116 (#34797)*, December 1979

[Coo72] D. C. Cooper, "Theorem Proving in Arithmetic Without Multiplication" *Machine Intellegence # 7*, 1972, pp. 91-99

[CRi81] L. Clarke and D. Richardson, "Symbolic Evaluation Methods for Program Analysis", from *Program Flow Analysis, Theory and Applications*, 1981, Prentice Hall

[CPa82] K. Cil and J. Pachl, "Folding and Unrolling Systolic Arrays" *University of Waterloo Research Report CS-82-11*, April 1982

[Den75] J. Dennis, "First Version of a Data Flow Procedure Language" *Project MAC, MIT*, May 1975

[DoD83] Department of Defense, "Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy" *ACM-SIGSOFT Engineering Notes, Vol 8 # 2*, April 1983, pp. 56–84

[Edw78] N. Edwards, "Configurable Pipelined Application Logic Systems" *IBM Research Report # RC 7313 (#31451)*, September 1978

[Fic83] Faith E. Fich, "New Bounds for Parallel Prefix Circuits" *Proceedings of the 15th ACM Symposium on Theory of Computing, pp. 100-109*, 1983

[FPa80] M. Fischer and M. Paterson, "Optimal Tree Layout" *University of Washington Tech Report 80-03-02*, February 1980

[FiR71] M. Fischer and M. Rabin, "Super-Exponential Complexity of Presburger Arithmetic" *Mit Tech Report MAC-TM-43*, 1971

[GPa83] Z. Galil and W. Paul, "An Efficient General-Purpose Parallel Computer" *Journal of the ACM, vol. 30 #2, pp. 360-387*, April 1983

[Gal80] C. Galtieri, "Architecture for a Consistent Decentralized System" *IBM Research Report #RJ2846(36132)*, June 1980

[GCP81] Cordell Green, Daniel Chapiro, and Thomas Pressburger, "Research on Synthesis of Concurrent Computing Systems", 1981, *Kestrel Tech Report*

[GKT79] L. J. Guibas, H. T. Kung and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms" *Procedings of the Caltech Conference on VLSI*, January 1979

[Gre81] Cordell Green, *et. al.*, "Research on Knowledge-Based Programming and Algorithm Design – 1981", 1981, *Kestrel Tech Report # KES.U.81.2*

[Gri75] P. Griffiths, "SYNVER: An Automatic System for the Synthesis and Verification of Synchronous Processes" *Harvard PhD Thesis and Tech Report TR-20-75*, June 1975

[Hac75] M. Hack, "Decidability Questions for Petri Nets" *PhD Thesis, MIT MAC Tech Report MAC-TR-161*, December 1975

[Hai81] B. Hailpern, "Modular Verification of Concurrent Programs" IBM Research Report #RC 9130 (#39971), *November 1981*

[Hal78] R. Halstead Jr., "Multiple-Processor Implementations of Message-Passing Systems", Masters Thesis, *MIT Tech Report MIT-LCS-TR-198*, January 1978

[Har80] S. Harbison, "A Computer Architecture for the Dynamic Optimization of High-Level Language Programs" *PhD Thesis, CMU, Tech Report CMU-CS-80-143*, September 1980

[Hil81] W. Hillis, "The Connection Machine (Computer Architecture for the New Wave)" *MIT AI Memo 646*, September 1981

[HMS83] P. Hochschild, E. W. Mayr, and A. Siegel, "Techniques for Solving Graph Problems in Parallel Environments" *Proceedings of the 24th Symposium on Foundations of Computer Science*, November 1983

[HMS84] P. Hochschild, E. W. Mayr, and A. Siegel, "Parallel Graph Algorithms" *Stanford Tech Report STAN-CS-84-1028*, December 1984

[Hoa78] C. A. R. Hoare, "Communicating Sequential Processes" *Communications of the ACM Vol. 21 # 8*, August 1978, pp. 666–677

[Hwa79] K. Hwang, "Computer Arithmetic; Principles, Architecture and Design", John Wiley & Co., *1979*

[Kan79] Elaine Kant, "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach", *Stanford Ph. D. Thesis*, 1979

[Ken81] K. Kennedy, "A Survey of Data Flow Analysis Techniques", from *Program Flow Analysis, Theory and Applications*, 1981, Prentice Hall

[Kin82] R. King, "Synthesis of Concurrent Computing Systems", Kestrel Tech Report # KES.U.82.10, *1982*, Chapter 1

[Kin83] R. King, "Research on Synthesis of Concurrent Computing Systems" *Proceedings of the 10th Symposium on Computer Architecture, pp. 39-46*, 1983

[KiB83] R. King and T. Brown, "Proposal for Research On Automatic Synthesis of Tree-Structured Concurrent Computing Systems", *Kestrel Tech Report #KES.L.83.1*, 1983

[Knu69] Donald Knuth, "The Art of Computer Programming"; Volume 2, Addison Wesley, 1969

[Knu73] Donald Knuth, "The Art of Computer Programming"; Volume 3, Addison Wesley, 1973

[Kun76] H. T. Kung and Charles E. Leiserson, "Systolic Arrays for VLSI", Sparse Matrix Proceedings, 1978

[KLe79] H. Kung and P. Lehman, "Systolic (VLSI) Arrays for Relational Database Operations" *Carnegie Mellon University Tech Report CMU-CS-80-114*, October 1979

[KuL76] H. T. Kung and Charles E. Leiserson, "Systolic Arrays for VLSI" *Sparse Matrix Proceedings, 1978*

[LFi80] R. Ladner and M. Fischer, "Parallel Prefix Computation" *Journal of the ACM, vol. 27 #4, pp. 831-838*, 1980

[Lap80] A. LaPaugh, "Algorithms for Integrated Circuit Layout: An Analytic Approach" *MIT Tech Report MIT-LCS-TR-248*, August 1980

[Lei81] F. T. Leighton, "A Layout Strategy for VLSI Which is Provably Good" *Proceedings of the 14th ACM Symposium on Theory of Computing, pp. 85-97*, 1982

[Lei84] F. T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting" *Proceedings of the 16th Annual Symposium on the Theory of Computing*, 1984

[LSa81] C. Leiserson and J. Saxe, "Optimizing Synchronous Systems" *Proceedings of the 22nd Annual Symposium on the Foundations of Computer Science, pp. 23-36*, 1981

[Len82] C. Lengauer, "A Methodology for Programming with Concurrency" *U. of Toronto Tech Report CSRG-142*, April 1982

[LiV81] Richard J. Lipton and Jacobo Valdes, "Census Functions: an Approach to VLSI Upper Bounds", *IEEE Symposium on the Foundations of Computer Science, 1981*pp. 13-22

[MaA84] R. Anderson and E. Mayr, "Parallelism and Greedy Algorithms" *Stanford University Tech Report STAN-CS-84-1003*,April 1984

[MJu83] D. McBride and R. Juels, "Directed Graphs for VLSI High-Level Synthesis" *IBM Research Report RC 9842 # 43417*, January 1983

[MeC80] C. Mead and L. Conway, "Introduction to VLSI Systems" *Addison-Wesley*, 1980

[Mil78] R. Milner, "Algebras for Communicating Systems" *Tech Report, University of Edinburgh #CSR-25-78*, April 1978

[MWi84] W. Miranker and A. Winkler, "Spacetime Representation of Computational Structures" *Computing 32*, 1984 pp. 93-114

[Opp78] D. Oppen, "A $2^{2^{2^{2^n}}}$ Upper Bound on the Complexity of Presburger Arithmetic" *Journal of Computer and System Sciences 16*, 1978 pp. 323-332

[Pai79] R. Paige, "Expression Continuity and the Formal Differentiation of Algorithms" *Technical Report #15, Courant Institute, New York, pp. 269-658*, 1979

[Pai82] R. Paige, "Transformational Programming – Applications to Algorithms and Systems" *Technical Report DCS-TR-118, Rutgers University*, September 1982

[Ram75] J. Rambaugh, "A Parallel Asynchronous Computer Architecture for Data Flow Programs" PhD Thesis, MIT MAC Tech Report MAC-TR-150, *May 1975*

[Ram73] C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by timed Petri Nets" *PhD Thesis, MIT MAC Tech Report MAC-TR-120*, July 1973

[ReV82] J. Reif and L. Valiant, "A Logarithmic Time Sort for Linear Size Networks", *Harvard Tech Report #TR-13-82*, 1982

[Sch80] J. Schwartz, "Ultracomputers" *ACM TOPLAS, vol. 2 #4 pp. 484-521*, October 1980

[Smi83a] D. Smith, "Derived Preconditions and Their Use in Program Synthesis" *Tech Report, Naval Postgraduate School, Montery, CA 93940*, November 1983

[Smi83b] D. Smith, "Top-Down Synthesis of Simple Divide & Conquer Algorithms" *Tech Report, Naval Postgraduate School, Montery, CA 93940*, November 1983

[SPn81] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis", from *Program Flow Analysis, Theory and Applications*, 1981, Prentice Hall

[SSC82] Siskind, Southard and Crouch, "Generating Custom High-Performance VLSI Designs from Succinct Algorithmic Descriptions" *Proceedings of the Conference on Advanced Research in VLSI*, January 1982

[Sto77] J. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", MIT Press, 1977

[TAm83] N. Takahashi and M. Amamiya, "A Data Flow Processor Array System" *Proceedings of the 10th Symposium on Computer Architecture, pp. 243-250*, 1983

[The82] D. Theriault, "A Primer for the Act-1 Language" *MIT AI Lab Memo 672*, April 1982

[Wag83] R. Wagner, "The Boolean Vector Machine [BVM]" *Proceedings of the 10th Symposium on Computer Architecture, pp. 59-66*, 1983

[Vit82] Vitanyi, "Real-Time Simulation of Multicounters by Oblivious One-Tape Turing Machines" *Proceedings of the 14th Annual Symposium on Theory of Computing*, May 1982

[Wen79] K. Weng, "An Abstract Implementation for a Generalized Data Flow Language" PhD Thesis, MIT Tech Report MIT-LCS-TR-228, *May 1979*

[Wol82] P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications" *Stanford Tech Report STAN-CS-82-925*, August 1982

END

FILMED

3-86

DTIC