Programming Parallel Computers

K. Mani Chandy

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-88-16

Programming Parallel Computers

K. Mani Chandy
California Institute of Technology
August 2, 1988
Caltech-CS-TR-88-16

Abstract

This paper is from a keynote address to the IEEE International Conference on Computer Languages, October 9, 1988. Keynote addresses are expected to be provocative (and perhaps even entertaining), but not necessarily scholarly. The reader should be warned that this talk was prepared with these expectations in mind.

Parallel computers offer the potential of great speed at low cost. The promise of parallelism is limited by the ability to program parallel machines effectively. This paper explores the opportunities and the problems of parallel computing. Technological and economic trends are studied with a view towards determining where the field of parallel computing is going. An approach to parallel programming, called UNITY, is described. UNITY was developed by Jay Misra and myself, and is described in [Chandy]. Extensions to UNITY are discussed; these extensions were motivated by discussions with Chuck Seitz.

1 Introduction

The success of our research in parallel programming depends in part on our understanding of the milieu in which programming will be carried out in the next ten to twenty years. So, let us study the history of programming to discern trends that may help us determine where the field is going.

1.1 Trends in Programming

Rapid changes in computer technology force us to confront, once again, the question: What is programming? When computers were first available, in the late 1940s, programming was the development of a program for a specific machine — the one in the basement of the laboratory in which the programmer was working. Computing resources were limited, and programmers had to go through whatever contortions were necessary to eke out the desired efficiency. Later, in the 1950s, the programmer's target computer became more general; it was no longer the machine in the basement, but a specific model: Model X of vendor Y. Later, in the 1960s, transportability became more important. Programs written for one model were required to run on others. Programs that were tailor-made to run efficiently on one model were difficult to transport to other models. The economics of the situation suggested that it was preferable to program for an abstract machine — a FORTRAN machine or a COBOL machine — and to pay the price of lower efficiency in return for the advantages of more expeditious program development, lower maintenance costs, and improved transportability. In the 1970s, the cost of software development and maintenance outstripped the cost of hardware. Programmers searched for higher-level languages to express programs accurately and succinctly, again at a potential cost of some loss in efficiency; Functional and Logic Programming were proposed as ways of handling the software development problem.

This little history shows a few clear trends. Programmers view computers in increasingly abstract terms — from that machine in the basement to an interpreter of functions. History shows a continuing shift in emphasis from concern about machine details to concern about expeditious program development.

1.2 Trends in Hardware

A computer has four basic parts: (a) processor, (b) storage, (c) communication channels, (d) interface. The relative improvements in technology in these four areas has a profound effect on programming because the important resource — i.e., the scarce resource — depends on the relative costs of these four parts. Efficiency in programming is concerned with using the

scarce resource most effectively; programmers can afford to be relatively profligate with abundant resources.

In the early history of programming, the processor was considered to be the scarce resource, and the emphasis was on using the processor most effectively. Rapid developments in silicon technology, and the relatively slower progress in magnetic storage technology have made storage the scarce resource. Consider the following facts:

Cost The cost of storage and communications exceeds the cost of procssors in many data-processing organizations.

Time The life-cycle of a transaction is roughly as follows: It spends some time getting processed, then it performs some I/O activity such as swapping, then more processing, then I/O activity such as paging or access to files, and so on ... , alternately using a processing unit and the I/O facility. The time a transaction spends doing I/O exceeds the time it spends using processing units. Even for low-priority 'batch' jobs, the time spent doing I/O usually exceeds the time spent *using* processing units (though the time spent *waiting* for processing units may be substantial).

Space The amount of silicon area required to implement a processor is the same as the amount required to implement a relatively small amount of memory — 2K to 10K bytes.

Programmers now work in a world in which processors are a relatively abundant resource, whereas storage and communication resources are relatively scarce. Indeed, the mismatch between a fast processing rate and a relatively slow memory access rate is not new; caches, operating systems that use extended storage, and semiconductor 'disks' are attempts at overcoming the relative difference in speeds between processing and data retrieval.

Developments in different technologies can change the picture dramatically; what was once a scarce resource may become abundant. For instance, the use of light as a medium of communication may make problems of interconnectivity much simpler, thus shifting the bottleneck to some other resource. Nevertheless, we should determine hardware trends as best we

can before embarking on programming research, because hardware technologies determine the scarce resource, and efficient programs are those that use scarce resources effectively.

1.3 Trends in Applications

There are a few trends in applications with important consequences for programming.

> The variety of applications has been growing dramatically. Applications range from scientific programs, where the primary data structures are arrays and grids, to expert systems that are based on sets of rules and list structures.
>
> The sizes of applications have increased; yet, progress in some fields of science continues to be limited by the size of problems that can be solved on computers within reasonable time.
>
> The lifetimes of successful applications are long and, in many cases, are getting longer.

The increasing variety of applications has resulted in the discipline of programming becoming fractured into diverse subdisciplines. It is a challenge to provide a uniform design methodology that applies across a variety of application areas.

The increase in the sizes and lifetimes of applications has resulted in a continuing concern about efficiency. In some areas (for example, many fields of scientific computation), an obsession with efficiency results in programmers designing programs from the very beginning so as to exploit special features of target machines — much as they once designed programs for machines in their basements. This obsession with efficiency is unavoidable given the limits that computation places in many areas of scientific research. The challenge is to mediate between two opposing concerns: On the one hand, the time taken to develop a correct program is smaller if programmers can use constructs that are close to their applications, and, on the other hand, program execution times are smaller if programs are written in notations that are designed to exploit scarce computing resources.

1.4 Trends in the Software Life Cycle

An elegant (but incorrect) picture of programming is as follows: The programmer is given a specification, and the programmer develops a program that meets the specification. The picture is incorrect because a large part of the cost of software development is in (a) coming up with the specifications in the first place, and (b) modifying the specifications and programs to meet changing demands. It is difficult to propose specifications because it is difficult to determine what customers want. Rapid prototyping — the rapid development of correct, but perhaps inefficient programs — is one way of handling this problem. Customers are given the inefficient prototype to determine if it has the functionality and the interface that is needed; if it does not, then the prototype is modified until it has the desired functionality. The prototype is used as the specification except for additional constraints regarding efficiency.

Specifications change with time, and modifying programs rapidly to meet changes in specifications is one of the challenges of programing.

In addition, rapid prototyping is useful for model building. In many areas of science, researchers build computational models of physical or sociological systems. The problem of model-building has two phases:

> Developing an appropriate model which may take repeated iterations of constructing a model and then determining whether it is an adequate representation of the system, and
>
> Improving the efficiency of model execution so that it runs rapidly enough to meet the researcher's needs for analyses of several sets of data.

Developing appropriate models is much the same as rapid prototyping, and improving model efficiency is a process of program refinement.

These trends suggest that a program design strategy should facilitate the development of rapid prototypes that can be transformed later, by stepwise refinement, into programs with adequate efficiency.

1.5 Proliferation of Architectures

From the first computer to the present, programmers have had to deal with one architecture: the von Neumann machine. Now there are many kinds of computing engines available: multicomputers with hundreds of processors that communicate by messages, SIMD machines with thousands of processors executing the same instruction, pipelined scientific processors, and neural networks. How can we respond to the challenge of a variety of novel architectures? One approach is to think of programming as the task of developing a program for a specific architecture. Thus, we have different programming disciplines: one for SIMD machines, one for message-passing MIMD machines, one for shared-memory MIMD machines, one for neural network machines, ... But, our study of the history of computing suggests that a trend of computing is to program for increasingly abstract (and general) computers. Just as programming Model X of vendor Y gave way to programming a FORTRAN machine, so too, programming for a specific architecture must give way to a more general view of programming. Programs will outlive the architectures for which they were initially designed, just as they outlived Model X; issues of transportability across *models* and *vendors* must be generalized to issues of transportability across *architectures*.

1.6 Learning from the Trends

The trends outlined in the previous section suggest a programming notation and design strategy based on the following ideas:

1.6.1 What, not When, Where or How

In the initial stages of design, a program consists of a description of *what* should be done; if the efficiency of the initial design is inadequate, programmers begin to address issues of *when* an action should be carried out, *where* (i.e., on which processor) an action should be carried out, and *how* a processor should carry out an action. The programming notation should allow programmers to describe programs exclusively in terms of *what*, and later to introduce concerns about *where*, *when*, and *how*. The notation should allow programmers to begin their designs by concentrating on the applica-

tion, and later to consider target architectures. Improvements in compiler technology will make it increasingly likely that the initial programs will have adequate efficiency.

1.6.2 Stepwise Refinement

If the efficiency of the initial program is inadequate, the notation should allow the program to be modified, in small steps, so that it becomes more efficient for a target architecture. One approach is to use one programming paradigm — say logic programming — at the initial stages of design, and then use a totally different paradigm — say imperative programming — when efficiency becomes a key concern. I think that there are advantages to using the *same* framework at all steps of the design, from the initial prototype to the final program. Therefore, the UNITY project is concerned with exploring the question: Will one framework suffice?

1.6.3 Assignment

The assignment operator is an effective way of managing memory. While there are several ugly things about the assignment operator, it does give the programmer control over how memory is used. If memory is a scarce resource, programmers need the power of the assignment operator. By the same argument, since data movement is often the bottleneck, programmers need the power of message-sending and message-receiving operators. And, since all computers today employ control flow and program counters, programmers need the power of sequencing. The challenge for a program design methodolgy is to allow programmers to use those constructs — equations, relations, fixed-points, invariants — they find most convenient to solve their problems (without being too concerned about efficiency) in the initial stages of design, and to later control the use of scarce resources — storage and communications — to obtain efficient programs.

1.6.4 Learning from Programming Paradigms

For many problems, logic programming appears to be the notation of choice; for many other problems, functional programming appears to be

ideal; for yet other problems rule-based programming is the style of choice. A difficulty with these approaches is that it is hard to refine programs down to the level of using the primitive hardware while remaining within these programming styles. The hardware — functional units, registers, caches, I/O devices — can be thought of as state-transition systems, but it is harder to represent them as functions or Horn clauses. The challenge is to use ideas from functional, logic and rule-based programming, while retaining the ability to bring in architectural considerations at appropriate stages in design. On the one hand, concepts from different programming styles appear to be useful; on the other hand, if we succumb to the temptation of including something from every programming style we will end up with a programming methodology that does not have a clean theory. Our challenge is to provide a unifying theory.

2 An Approach to Programming

The ideas here are from UNITY, a programming methodology, including a specification notation, programming notation, proof system and design heuristics, that was proposed in [Chandy] which contains both an informal discussion of the key ideas and a formal treatment. The discussion in here, however, is entirely informal. First we shall describe the ideas underlying UNITY; later we shall discuss possible extensions motivated by discussions with Prof. Chuck Seitz of Caltech.

A UNITY program has four parts:

declare section A declaration of variables.

always section A *set* of equations that are invariants of the program, i.e., that always hold in all computations.

initially section A *set* of equations specifying the initial values of variables.

assign section A *set* of assignments.

(The ordering of equations or assignments in each section is irrelevant.)

An assignment has one or more variables on the left-hand side and a corresponding number of expressions on the right-hand side. The syntax of

an equation in a UNITY program is the same as that used for an assignment statement, except that the assignment ':=' in an assignment statement is replaced by the equality '=' in an equation. When executing an assignment statement, all expressions on the right-hand side are evaluated concurrently and then applied to the left-hand sides concurrently. Thus, $x, y := y, x$ interchanges the values of x and y. An equation with more than one variable on the left-hand side is equivalent to a set of equations, one for each variable on the left side. Thus: $x, y = 1, 2$ is equivalent to: $x = 1, \quad y = 2$

Some of the sections of the program may be empty. We begin by describing programs in which the last two sections, the *initially* and *assign* sections, are empty. Such programs are functional programs. Functional programs consist of a set of equations (definitions); the solution to the set of equations is the result of the program. Definitions are given in the *always* section, and variables are declared in the *declare* section.

2.1 Functional Programs

Consider the problem of sorting an array in increasing order. We are given an array, X, each element of which is distinct. Extensions to allow duplicates are straightforward. We are required to design a procedure that returns an array Y, where (a) Y is a permutation of X and (b) Y is in increasing order. Assume that both X and Y are indexed $1 \ldots N$.

2.2 First Program

Let *perm* be a function where $perm(X)$ is the set of all permutations of X. One solution is:

Define Y as the lexicographically smallest member of $perm(X)$.

(An array Z is lexicographically smaller than an array Z' of the same

dimension as Z if and only if, for some index i, $Z[i] < Z'[i]$, and for all indices j less than i, $Z[j] = Z'[j]$. Thus $[2,1,3]$ is lexicographically smaller than $[2,3,1]$.)

The program is a declaration and an *always* section that consists of the equation that defines Y.

Given function *perm* and operand *lexmin* — the lexicographical-minimum dyadic operand — the definition of *sort* is simple, and the correctness arguments are straightforward. The execution time (which is $N!$ in a straightforward implementation) may be unacceptable.

2.2.1 Notation

The program consists of a declaration of Y and an *always* section that consists of the single equation:

$$Y = \ll lexmin\ Z\ :\ Z\ in\ perm(X)\ ::\ Z\ \gg$$

In general,

$$\ll\ op\ Z\ :\ Z\ in\ S\ and\ f(Z)\ ::\ g(Z)\ \gg$$

where *op* is an associative, commutative, dyadic operator, and where f is a boolean function, has value

$$g(Z_1)\ op\ g(Z_2)\ op\ g(Z_3)\ op\ \ldots\ op\ g(Z_k)$$

where $Z_1, Z_2, Z_3, \ldots, Z_k$ are the members of S that satisfy f.

2.3 Second Program — The Rank Sort

Next, we describe another functional program.

Define an array rank of integers indexed $1 \ldots N$ *where for all* i: $rank[i]$ *is the number of elements of* X *that are less than or equal to* $X[i]$. *Since* $rank[i]$ *is the position of* $X[i]$ *in array* Y, *it follows that* $Y[rank[i]] = X[i]$.

For example, given that $X = [3, 7, 5]$, $rank = [1, 3, 2]$. So, in our example, $Y[1] = 3$, $Y[3] = 7$, and $Y[2] = 5$.

The program has a *declare* section and an *always* section that consists of equations that define *rank* and Y.

2.3.1 Notation

The *always* section for this program is:

always
{The equations defining *rank* are: }
$\ll \Box i \ : 1 \leq i \leq N \ :: rank[i] =$
$\qquad \ll +j \ : (1 \leq j \leq N) \wedge (X[j] \leq X[i]) \ :: 1 \gg$
$\gg$
$\Box$
{The equations defining Y are: }
$\ll \Box i \ : 1 \leq i \leq N \ :: Y[rank[i]] = X[i] \gg$

The symbol $\Box$ separates equations. There is one equation defining $rank[i]$ for each i in $1 \ldots N$. The term

$$\ll +j \ : (1 \leq j \leq N) \wedge (X[j] \leq X[i]) \ :: 1 \gg$$

in the definition of $rank[i]$ is a sum (because of the '+') over all j, such

that $(1 \leq j \leq N) \wedge (X[j] \leq X[i])$, of the expression '1'. Thus, it is a count of the number of elements in X that are less than or equal to $X[i]$. There is one equation defining $Y[rank[i]]$ for each i in $1 \ldots N$.

2.4 Other Functional Programs

There are many other functional definitions of sorting. We do not discuss them here because our modest goal is only to present some idea of UNITY. The message I want to leave with you is this: A UNITY program that consists of only the *declare* and *always* sections is a functional program, and functional programs are often clear and succinct.

3 Rule-Based Programs

Now let us consider programs without the *always* section but with the *initially* and *assign* sections. These programs are similar to rule-based programs.

Computation proceeds by executing an assignment selected nondeterministically and fairly from the set of assignments. The fairness requirement is that it is always the case that each assignment will be executed eventually. The state of a program is given by the values of its variables. (There is no program counter because there is no sequential control flow.) A fixed point of a program is a state of the program in which the execution of each assignment leaves the state unchanged; therefore a fixed point of a program is a state in which, for each assignment in the set, the values of variables on the left-hand side of the assignment equals the values of the corresponding expressions on the right-hand side. The program returns values if and when it reaches a fixed point, and, in this sense, reaching a fixed point is similar to termination. (Of course, many programs — operating systems, for instance — never terminate, and do not reach fixed points; they may become quiescent awaiting input, but they do not halt. In this paper we shall restrict attention to programs that are required to reach fixed points.)

The set of assignments can be thought of as a set of actions or a set of rules. A program containing the *assign* section has points of similarity with

rule-based programs. There are also similarities between sets of assignments and sets of communicating sequential processes and sets of objects; we do not have space here to explore these similarities.

3.1 Third Program — An In-Place Sort

Next, we design an in-place sort. Initially $Y = X$. We define *what* actions are to be carried out on Y without specifying *where, when or how* the actions are to be carried out; therefore, the program is simple.

Initially $Y = X$.

Rules *For every adjacent pair of elements of* Y*: flip the pair if they are out of order and leave them unchanged otherwise. (The program has* $N - 1$ *rules, one for each adjacent pair of elements.)*

The program consists of an *initially* section in which Y is set equal to X, and an *assign* section which consists of assignments corresponding to the actions of flipping adjacent pairs of elements of Y if they are out of order. Computation proceeds by repeatedly selecting any adjacent pair of elements nondeterministically and fairly, and flipping them if they are out of order. A fixed point of the program is a value of Y in which flipping an adjacent pair of elements of Y *if they are out of order* (and leaving them unchanged if they are in order) leaves Y unchanged. To demonstrate the correctness of the program, we have to show that (1) a fixed point will be reached, and that (2) at all fixed points reached in computations of the programs, Y is a permutation of X, and Y is in increasing order. More about correctness next.

3.1.1 Correctness Arguments

Metrics and Progress To prove that all computations of the program will reach a fixed point, we demonstrate a *metric* — a function from the states of the program to the integers, where the value of the function is bounded from below and every state change reduces the value of the function. If the value of the metric is m initially, and a lower bound on the metric is b, a fixed point is reached in at most $m - b$ state changes. For our example, a metric for the sorting program is the number of out-of-order pairs (i, j); i.e., it is the number of pairs (i, j) where $i < j$ and $Y[i] > Y[j]$. We prove that a function, M, from the states of a program to the natural numbers, is a metric as follows. We show that for each assignment, and for all values Y of the state that satisfy invariants of the program, execution of the assignment in state Y results in a state, $newY$, where

either there is no change in state, (i.e., $Y = newY$), or

the metric decreases (i.e., $M(Y) > M(newY)$).

For our example, our proof obligation is to show that flipping an out-of-order pair reduces the total number of out-of-order pairs. This is straightforward.

Demonstrating a bound on the number of state changes is not the only way of proving that a fixed point will be reached. In general, all we have to show is that a metric will decrease *eventually*, and we do not have to give a bound for how many state changes will occur before that eventuality is realised.

Invariants An invariant of a program is a predicate on the states of the program that holds in all states reached in all computations. In particular, an invariant holds at all fixed points reached in computations of the program. To prove that a predicate Inv on the states of a program is an invariant of the program, we prove that Inv holds initially, and we also prove that for each assignment in the program if Inv holds immediately before the assignment is executed, then it holds immediately after the assignment is executed.

In our example, we shall prove the following invariant:

Y is a permutation of X.

Initially, Y is a permutation of X because $Y = X$. If Y is a permutation of X immediately before a pair of elements of Y is flipped, Y will remain a permutation of X immediately after the flip. This completes the proof of the invariant.

Fixed Point A fixed point of the program is a state in which each action — flipping a pair of adjacent elements of Y if they are out of order, and leaving them unchanged if they are in order — leaves the state unchanged. If Y is not ordered, flipping adjacent elements that are out of order changes Y. Therefore, Y is ordered at all fixed-points of the program.

Correctness of the Program We have demonstrated a metric; therefore, the program reaches a fixed point. We have shown that Y is ordered at all fixed points of the program. We have shown that an invariant of the program is that Y is a permutation of X. Therefore, the program reaches a fixed point in which (1) Y is ordered and (2) Y is a permutation of X.

3.1.2 Notation

The action of switching $X[i]$ and $X[i+1]$ if they are out of order is represented by the multiple assignment:

$$X[i], X[i+1] := min(X[i], X[i+1]), max(X[i], X[i+1])$$

In this assignment, $X[i]$ becomes the smaller of $X[i], X[i+1]$, and, *simultaneously*, $X[i+1]$ becomes the larger of $X[i], X[i+1]$. The program consists of $N-1$ actions, one for each i, where $0 < i < N$. The set of $N-1$ actions is represented by:

$\ll \Box i : 0 < i < N :: X[i], X[i+1] :=$
$\qquad min(X[i], X[i+1]), max(X[i], X[i+1]) \gg$

The box $\Box$ is a separator between assignments. In the above statement, i is a dummy variable, and the program has assignments for all i, where i satisfies the predicate between the ':' and the '::', namely $0 < i < N$.

4 Programs with Functional and Rule-Based Components

A variable that is defined in the *always* section cannot be assigned a value in the *assign* section; it may, however, appear on the right-hand side of an assignment. For example, let nf and nm be program variables that are the numbers of females and males (respectively) in an organization. The total number of people, nt, in the organization can be defined by the invariant: $nt = nf + nm$. The equation defining nt appears in the *always* section. Variables nf and nm can appear in the left-hand sides of assignment statements, but nt cannot. The program is equivalent to one in which all occurrences of nt in the right-hand sides of assignments are replaced by the expression $(nf + nm)$ that defines it.

4.1 Fourth Program — Towards the Heap Sort

We next describe another in-place sort. This program can be refined to obtain the heap sort, an efficient program for sequential architectures. Let m be an integer variable that takes on values in $1 \ldots N$. The program has the invariant that the portion $Y[m+1 \ldots N]$ of the array has its final values, i.e., for all k where $m < k \leq N$: $Y[k]$ is the k-th smallest element of X.

Define Define *index, big, such that* $Y[big]$ *is the largest value in* $Y[1 \ldots m]$.

Initially $Y = X$ *and* $m = N$.

Rules *The program has a single rule: Interchange* $Y[m]$ *and* $Y[big]$ *and concurrently decrement* m *if* $m > 1$.

Define *big* in the *always* section. The *assign* section consists of a single rule.

I do not contend that the UNITY program is better, more succinct or clearer than an imperative program. I do contend, however, that a UNITY program and a typical imperative program emphasise different things.

In UNITY, one tends to use definitions where possible, and to avoid control flow. For instance, we could have modified the value of *big* in the set of rules; we prefer to define *big* by means of an equation (that always holds). An argument can be made, however, that control flow has been introduced by the *always* section. For instance, in our example of a sorting program, *after* each execution of the rule (that interchanges $Y[big]$ and $Y[m]$, and decrements m) the value of *big* is recomputed. This control flow is *implicit* in a UNITY program, whereas it is *explicit* in imperative programs. Indeed, in UNITY, we choose not to think about operations at all, except to evaluate efficiency.

Also, where one would write a loop in an imperative program, one tends to define a set of variables and write a *single* assignment statement in UNITY. Often, the assignment statement assigns multiple values concurrently to multiple variables. Sequencing is not necessarily bad, nor is concurrent assignment necessarily good. The difference in emphasis does, however, lead to different styles of program development: In an imperative program, one tends to design the program skeleton and its proof together, whereas in UNITY one tends to design the specification in detail without proposing a program skeleton in the usual sense. In this example, we began by giving an invariant for our program. In a sequential program, one would propose a program skeleton and annotate the program skeleton: different assertions hold at different points in the program. Since UNITY programs do not have 'program points', we can propose an invariant without ambiguity about *where* the invariant holds: The invariant holds for the entire program.

4.2 Notation

Let v be the largest element in $Y[1 \ldots m]$. Then *big* is any index such that $Y[big] = v$; we define *big* to be the smallest index such that $Y[big] = v$.

Program *P4*

declare $big, v : integer$;

$X, Y : array[1 \ldots N]\ of\ integer$

always $v = \ll max\ i\ :\ 1 \leq i \leq m\ ::\ Y[i]\ \gg$

$\Box\, big = \ll min\ i\ :\ (1 \leq i \leq m) \wedge (v = Y[i])\ ::\ i \gg$

initially $\ll \Box\ i\ :\ 1 \leq i \leq m\ ::\ Y[i] = X[i] \gg$

$\Box\ m = N$

assign $Y[m], Y[big], m := Y[big], Y[m], m - 1 \qquad if\ \ m > 1$

end $\{P4\}$

5 Refinement for Target Architectures

A first step in design is to develop a prototype quickly. The prototype is developed without too much concern for efficiency or target architectures. Later in the design cycle, the prototype is refined to execute efficiently on a set of target machines. We do not have space here to carry out the refinement of sorting programs for different architectures. The rank sort is efficient for parallel machines. The fourth program can be refined to obtain the heap sort. In addition, we shall also present another example of an in-place sort — the odd-even transposition sort — that appears to be efficient for both SIMD and MIMD architectures.

Introduce two sentinels, $Y[0]$ and $Y[N+1]$, where $Y[0] = -\infty$ and $Y[N+1] = +\infty$. Let k be the step number, i.e., the number of times that a rule has been executed.

Initially $Y = X$ *and* $k = 0$.
The program has a single rule: flip $Y[i]$ *and* $Y[i+1]$ *if they are out of order, for all* i *where* $(i - k)$ *mod* 2, *and concurrently increment* k *by* 1 *if* $k < N$.

Pairs $(Y[1], Y[2]), (Y[3], Y[4]), (Y[5], Y[6]), \ldots$, are flipped, if they are out of order, on odd steps. Pairs $(Y[2], Y[3]), (Y[4], Y[5]), (Y[6], Y[7]), \ldots$, are flipped, if they are out of order, on even steps. After N steps, the rule does not change the state of the program and, therefore, a fixed point is reached.

5.1 Notation

The *assign* section of the program is:

$$\textbf{assign} \ll || \ i \ : 1 \leq i \leq N \ :: Y[i] := \begin{array}{l} min(Y[i], Y[i+1]) \ if \ (i = k) \ mod \ 2 \sim \\ max(Y[i], Y[i-1]) \ if \ (i \neq k) \ mod \ 2 \end{array}$$
$$\gg$$
$$|| \qquad k := k + 1 \ if \ k < N$$

The '||'operator is a parallel assignment. The right-hand sides of the assignment are evaluated in parallel for all i, and then assigned in parallel to $Y[i]$ for all i.

Mapping this program to SIMD machines, such as the Connection Machine, is fairly straightforward. Mapping the program to multicomputers and shared-memory multiprocessors is also quite direct; for details, see [Chandy].

6 Extensions

In this section we shall propose two extensions to UNITY that make rapid prototyping easier, and also suggest a way towards stepwise refine-

ment for parallel architectures. These extensions control the manner in which assignments in the *assign* section are executed. The extensions were motivated by discussions with Chuck Seitz. We shall discuss one extension at a time. Each extension is an additional section of a UNITY program.

6.1 Guarantee

The first extension is called a *guarantee* section which consists of a single boolean expression on the variables of the program and a variable $new.y$ for each variable y declared in the program. Let $g(y, new.y)$ be the boolean expression. Computation progresses by selecting an assignment nondeterministically and fairly (as before); the selected assignment, say $y := e$, is executed only if $g(y, e)$ holds; otherwise the assignment is skipped. Therefore, the program with the *guarantee* is equivalent to one without the *guarantee* in which each assignment $y := e$ is replaced by an assignment $y := e \; if \; g(y, e)$. So the *guarantee* does not provide additional power. The reason for the *guarantee* is:

to focus attention on properties of the program rather than on properties of individual statements.

Pseudovariable, $new.y$, is the next value of y if the selected assignment is executed; thus $g(y, new.y)$ relates the current value of y with its next value.

Let us consider the sorting example again. We propose to do a sort in place; therefore, initially $Y = X$. We begin, as usual, by determining *what* the actions of our program should be. Let us say that the actions of the program are to interchange *any* pair of elements of Y. Interchanging an arbitrary pair of elements will maintain the invariant that Y is a permutation of X, but it will not result in the computation progressing towards a result. Therefore we propose the *guarantee* that a metric decreases, where the metric is defined as the total number of out-of-order pairs. Thus, a pair of elements is interchanged only if the number of out-of-order pairs is decreased. Specifying a metric in this manner is one way of guaranteeing that the program will reach a fixed point. It makes programming simpler by simplifying the correctness argument; programmers do not have to prove that fixed-points will be reached — they need only specify a metric in the *guarantee* section. (Of course, the metric must be bounded from below.)

We can also employ the *guarantee* section to define invariants of a program. For example, suppose we want $x > y$ to be an invariant of our program. Then we define the *guarantee* so that it implies $new.x > new.y$, and we define the *initially* section so that $x > y$ initially. An assignment (that is selected nondeterministicaly and fairly, in the usual manner) is executed only if the desired invariant is maintained; otherwise, the assignment is skipped.

Here, again, the purpose of the *guarantee* is to simplify the correctness argument — the programmer need not prove an invariant; it is enough to put it in the *guarantee* section.

At a fixed point of a program, for each assignment $y := e$:

$$(y = e) \ \ or \ \ not\ g(y, e)$$

where g is the guarantee. For a program with an *assign* section, our correctness arguments usually have three parts:

demonstration of a metric,

proofs of invariants,

proofs of properties at fixed-points.

By employing the *guarantee* section the proof obligation can be limited to the fixed-point part. If the initial program does not have adequate efficiency, stepwise refinement is employed to transform the program by weakening the guarantee and strengthening the assignments.

In summary, I want to leave you with the following message about the *guarantee* section. A critical difference between a prototype that is developed rapidly and the final program is that the protype has a simpler proof — its correctness is more obvious. The simplicity in proof is often achieved at the expense of efficiency in execution. **The *guarantee* section is an aid to rapid prototyping because it simplifies the correctness argument.** The direction in which a program can be made more efficient is often self-evident from the *guarantee*: Weaken the guarantee and strengthen the assignments.

6.2 Goal

In this section, we restrict attention to rule-based programs, i.e., programs with *assign* sections. (These programs may have other sections as well.) Consider only programs for which it is possible to demonstrate a metric — a function from the states of the program to the natural numbers — such that every state change reduces the metric. If the initial value of the metric is M, then the program reaches a fixed point in at most M state changes.

Consider a computation tree for such a program. The nodes of the tree are states of the program; edges of the tree are labeled with rules (i.e., assignments in the *assign* section). Each node has N sons where N is the number of rules in the program. This tree has infinite depth. A computation of the program is a path through the tree, starting at the root, and going from each node to one of its sons (subject to the fairness constraint).

Derive a tree with a bounded number of nodes from the computation tree as follows: Discard each node that is identical to its father, i.e., retain in the tree only those rule-executions that change the program state. (When a node is discarded, all its descendants are discarded as well.) Since each state-change reduces the metric, and the metric has value M at the root, this tree has depth at most M. This tree has a bounded number of nodes since it has bounded depth and each node has at most N sons. Many problems can be formulated as a search of this tree.

Consider, for example, the eight-queens problem. The goal is to find a placement of eight queens on a chessboard so that no queen can capture another. Consider the following rule-based program.

Initially *Initially the board is empty.*

Actions *An action (or rule) is 'place a queen on an empty square of the board'. (There are 64 actions corresponding to the 64 locations on the board.)*

Guarantee *No queen on the board can capture another queen on the board.*

A metric for this program is $(8-n)$ where n is the number of queens on the board. We want our program to return any board position that satisfies the guarantee and has eight queens on it. For instance, we may design our program to return the 'least' such board where the least is any ordering (such as a lexicographic ordering).

We may be faced with the problem of counting the number of solutions to the eight-queens problem, or of listing all solutions in which there is a queen in row 1 and column 1. We specify what we want the program to do in the *goal* section. Let *state* be any reachable state of the program. We specify the goal by quantifying over all states in the usual way. For instance:

$$\textit{Goal} \;\; << \textit{least state} \;:\; 8 \textit{ queens on board} \;::\; \textit{board} >>$$

returns the least board over all reachable states of the program in which there are eight queens on the board.

$$\textit{Goal} \;\; << + \textit{state} \;:\; 8 \textit{ queens on board} \;::\; 1 >>$$

returns the number of reachable states in which there are eight queens on the board.

(Note: A board can be represented by a boolean array b where $b[r,c]$ holds if and only if the r-th row and c-th column of the board contains a queen. In the following, assume that b is indexed $[0\ldots7, 0\ldots7]$.)

A somewhat more efficient solution is as follows. For a given row r (which is initially 0) and for each of the eight columns c, place a queen in row r, column c, and increment r by 1 if $r < 8$. The search tree for this program is less wide than the tree for the previous program because each node of the tree has at most eight sons.

Initially *Initially the board is empty, and* $r = 0$.

Actions *For each column c: 'place a queen in row r, column c, and increment r by 1 if* $r < 8$.' *(There are 8 actions corresponding to the 8 columns.)*

Guarantee *No queen on the board can capture another queen on the board.*

Goal $<<$ *least state* : 8 *queens on board* :: *board* $>>$

We can continue the process of making our program more efficient by, firstly, reducing the size of the search tree, and later by considering operational issues such as the manner in which the program is implemented on a target architecture. Our objective is not to exclude operational and architectural considerations, but rather to include them only if efficiency demands it.

Consider another example: The knapsack problem. We are given a number of objects and a knapsack. Each object has a weight and value. The knapsack has a capacity. Objects are to be placed in the knapsack to maximize the total value of all the objects in the knapsack, subject to the constraint that the total weight of all the objects in the knapsack must not exceed the capacity of the knapsack. An obvious solution is as follows. An action is: 'place an object in the knapsack' (if it is not already in the knapsack). The guarantee is that the capacity of the knapsack is not

exceeded. The goal is to maximize the contents of the knapsack over all states:

Goal $<<$ *max state* : *true* :: *sum of values of objects in knapsack* $>>$

Another way of thinking about the *goal* section is in terms of fairness. In a program that does not have the *goal* section, the execution of rules in the *assign* section is fair: It is always the case that every rule will be executed eventually. This fairness rule is quite weak; we can construct an execution where the 'right' rules are not executed for an arbitrarily long (but finite) time, and the 'wrong' rules are executed repeatedly. In this sense, the nondeterminism in rule-execution is *demonic*. In the initial stages of design, we work with programs with nondeterministic execution sequences; as design progresses and the target architecture becomes better defined, we make our programs more deterministic by disallowing sequences that are inefficient for the target architecture.

Chuck Seitz suggested a complementary form of program development: Begin by assuming an *angelic* form of nondeterminism, in which the 'right' rules are always executed; later implement this angelic behavior on the target computer. The angelic and demonic views of nondeterminism are quite symmetric; they are helpful in program design for the same reason: We unnecessarily restrict program execution if we insist that programs have to be deterministic.

How do we specify the 'right' rules? We specify a *goal*; the 'right' rules are those that get the program to its goal. The state of the program is changed, as usual, by carrying out one of the actions. Rather than select actions demonically, however, we may think of the actions as being selected angelically so that the goal is achieved eventually.

Rule-based programs with angelic nondeterminism appear to have, at least superficially, some similarities to logic programming. (Logic programming is nicer in that it consists of a set of definitions rather than a set of actions.) We hope to use nondeterminism as a means of stepwise refinement, and both angelic and demonic nondeterminism seem useful in this regard.

6.3 Caveat

I do not know if the ideas of *goal* and *guarantee* are good. Jay and I write many programs and cogitate for a long time before we decide that a construct is worth including in UNITY. I think the ideas are nice, but then, I am often wrong.

7 Program Composition

UNITY can be thought of as an experiment in studying different forms of program composition. In place of an assignment, in the *assign* section, we can have a program — a procedure — that modifies some variables (corresponding to the variables on the left-hand side of the assignment) and that accesses other variables without modifying their values (corresponding to variables that appear on the right-hand side, but not on the left-hand side, of the assignment). A rule-based program in which (each rule is itself a program, and) rules are executed using demonic nondeterminism, and in which values are returned at fixed points, is one way of program composition. Similarly, a rule-based program using angelic nondeterminism is another means of program composition. A multiple assignment corresponds to another form of program composition: Replace each assignment by a program, and the multiple assignment corresponds to a composed program in which all of its component programs are executed 'in parallel'. Indeed, all UNITY constructs are about program composition. The only traditional form of program composition is sequential composition; UNITY offers some others. Here we are following Tony Hoare's work [Hoare], which can be thought of as a study of program composition. The programs composed by using UNITY constructs may be written in any notation — an imperative language, for instance.

8 Adequacy of System Representation

One of our goals is to refine our program down to a detailed level, 'close' to the hardware. We can only carry out such a refinement if the notation allows the hardware, or the underlying system, to be represented adequately.

The underlying system may be an asynchronous circuit [Martin,Seitz], or a clocked circuit, or a distributed system that spans continents. I do not have the time to discuss adequacy of representation in detail. I shall merely allude to it in passing.

First consider asynchronous circuits. A wire connecting the output variable y of a gate to the input variable z of a gate is represented by the assignment (or rule) $z := y$. A gate with a vector of inputs y, and a vector of outputs z, that is computing a function f, is represented by the assignment $z := f(y)$. For example, an *and* gate with inputs x, y and output z is represented by: $z := x \ and \ y$. Assignments are selected nondeterministically and fairly (demonically) as usual. Thus an asynchronous circuit is represented by a rule-based program in which each circuit and each wire is represented by a rule, and the representation is straightforward.

Now consider synchronous circuits. A multiple assignment is used to represent the firing of several gates in a single clock cycle. For example, consider the *and* gate of the previous paragraph and an *or* gate with inputs a and b and output c, where all the signals a, b, c, x, y, z are distinct. The outputs of the gates are read once in each clock cycle, after the voltages have reached equilibrium. The circuit is represented by:

$$z := x \ and \ y \ \ || \ c := a \ or \ b$$

Asynchronous and synchronous communication in distributed systems can also be represented in the same way. Our goal is to employ the same notation — and the same ways of thinking — at all levels of program design, and this requires that one model be an adequate representation at all levels. I most certainly do *not* wish to suggest that our representation is preferable to other representations of circuits or systems; I do, however, want to emphasise that adequacy of representation is an important issue, and it appears that UNITY offers some hope in this regard.

9 Success or Failure?

When embarking on a scientific experiment, it is a good idea to evaluate its potential for success or failure. Let me try to do that now with respect to the UNITY experiment. First, the potential for failure.

9.1 Failure

The UNITY experiment may fail because there may be no unity to the program design task: programming different architectures and different applications may be so different that a general approach will offer little insight.

The experiment may fail because UNITY constructs are unconventional. We have several decades of experience in imperative deterministic programming: We *know* that programmers can think in this style. I am far from certain that programmers will be willing to adopt the UNITY style. For example, the coupling of definitions (in the *always* section) and nondeterminsm (in the *assign* and *goal* sections) may be unacceptable to most programmers. Consider another example: Will programmers be comfortable thinking of a sort program merely as ' *flip out-of-order pairs*' ?

The experiment may fail because the programming model is more abstract than the conventional model, and so understanding UNITY programs requires a better understanding of proof constructs such as invariants, metrics and progress than is required for sequential programs.

UNITY is predicated on the assumption that a variety of architectures exist for a variety of reasons, and that a variety of programming styles exist for a variety of reasons; a goal is to attempt to provide unifying principles across architectures and notations. An alternate view (and possibly the correct view) is that architectures and notations are merely artifacts: They exist because we, the computing community, want them to exist. Therefore, we could all agree to employ the single best notation, and then build machines for only that notation; in this case the problem of unification vanishes.

Finally, UNITY may fail because its specification notation, programming constructs, proof method, and heuristics are just plain wrong. It will most definitely fail if, in our attempt to unify, we create a hodgepodge of ideas without unifying principles.

9.2 Success

UNITY is not a programming language: It is a collection of ideas about specifications, programming constructs, proofs, and heuristics. The effi-

ciency with which UNITY programs execute on multicomputers, SIMD machines, pipelined architectures and sequential machines, are certainly reasonable yardsticks for evaluating success. Our experiment, however, is not about a language. If the UNITY ideas are good, we expect that they will find their way (after some modification) to other languages and systems. For instance, if our ideas about nondeterminism, fixed points and rules are good then these ideas will find their way to into research that others do. I think it is unlikely that others will use our notation because notation seems to be a matter of personal style, but that does not matter. For instance, others may use guarded commands rather than assignments, and equilibrium states rather than fixed points. The key question is whether the ideas underlying the notation are similar. If our representation of circuits is good, the same representation will be used by others. If our proof method is good then it will lead to similar (and better) methods. Finally, if the UNITY experiment is a good one, others will try similar experiments. The success of the project should be measured by the degree of diffusion of UNITY ideas. Only time will tell whether the project is successful.

10 Conclusion

Our goal in program design is to allow programmers to focus attention initially on their problems rather than on the computer architectures on which their programs may be required to run. If the initial programs do not execute fast enough (and we hope that in many cases, with optimizing compilers and fast machines the initial programs will be adequate), then the programs are refined to make them execute faster. A program runs fast if it uses scarce resources well, even if it is profligate in its use of abundant resources. Our goal is to provide a path by which in the initial design stages programmers can use constructs such as equations, invariants and metrics, that do not deal with architectures or computing resources, and in later stages, they can control message communication, memory management and sequencing, and the special features of target architectures. For example, in the initial stages of designing an in-place sort program, our program might be: flip adjacent out-of-order pairs. A compiler can compile efficient code for parallel machines (SIMD, multicomputers and multiprocessors) for this

program; indeed, it is fairly direct to generate the odd-even transposition sort. If, however, the compiled code does not execute fast enough on the target machine, the programmer can refine the initial program by employing a set of heuristices to obtain shorter execution times.

A program design strategy includes a specification notation, a proof theory, and heuristics for the refinement of designs; we have no space to discuss these important issues, and the reader is referred to [Chandy]. We are currently writing compilers for SIMD machines. See [Bagrod] in this conference for plans for the Connection Machine.

11 Acknowledgments

The UNITY project, a research effort that has been going on for the last three years, is funded by the Office of Naval Research, partly through the University Research Initiative, and is led by Jay Misra at the University of Texas at Austin and me. Prof. Rajive Bagrodia of the University of California at Los Angeles is extending UNITY and implementing it on the Connection Machine. Eric F. Van de Velde of the California Institute of Technology is experimenting with extensions of UNITY for scientific computations executing on coarse-grained message-passing concurrent computers [Velde]. Their efforts continue to suggest research directions. Many of the ideas discussed here came from discussions with Chuck Seitz who, with his students, has developed notations and many algorithms for multicomputers. Special thanks to Alain Martin for his thoughtful comments.

Also, the support of the California Institute of Technology and the Sherman Fairchild Foundation during the year 1987 - 88 is gratefully acknowledged.

Many of the ideas here are drawn from different programming styles: communicating sequential processes, and object-oriented, rule-based, functional, logic and imperative programming. It is not possible to cite the sources from which these ideas are drawn in the space available. A complete list of references is found in [Chandy].

Bibliography

[Bagrod] R.Bagrodia and K. M. Chandy, 'Programming the Connection Machine', Proceedings IEEE International Conference on Computer Languages '88, Miami Beach, Florida, October 9 13, 1988.

[Chandy] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts, 1988.

[Hoare] C. A. R. Hoare *Communicating Sequential Processes*, Prentice-Hall, London, 1984.

[Martin] A. Martin, 'Compiling Communicating Processes into Delay-Insensitive VLSI Circuits', *Journal of Distributed Computing*, Vol.1, No.3, 1986,

[Seitz] C. Seitz, 'System Timing' in *Introduction to VLSI Systems*, eds. C.Mead and L.Conway, Addison-Wesley, Reading, Massachusetts, 1980.

[Velde] E. F. Van de Velde, 'A Concurrent Direct Solver for Sparse Unstructured Systems', report C3P-604, Caltech Concurrent Computation Project, 1988.