Derivation of Efficient Parallel Programs: An Example From Genetic Sequence Analysis

Ambuj K. Singh
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Ross Overbeek
Argonne National Laboratory
Argonne, Illinois 60439

Abstract

Implementation issues such as synchronization of shared data, implementation of abstract data types, and scheduling of processes are usually not addressed in the formal derivation of parallel programs. We seek to redress the situation by considering these issues in the context of developing an efficient implementation of an actual parallel program. The computational problem that we selected is motivated by work in aligning sequences of genetic material. We proceed by developing an algorithm in Unity and investigating the issues that arise in producing an efficient C implementation of the resulting algorithm. Along the way, we develop some theorems about program refinements, and illustrate the usefulness of the theorems in the context of refining the original Unity program.

This work was supported in part by ONR Grants N00014-86-K-0763 and N00014-87-K-0510 and in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

---

1 Introduction

The usefulness of program derivation through stepwise refinement has been demonstrated repeatedly [1, 2, 3, 4, 6]. However, for this approach to be widely used for writing efficient parallel programs, implementation issues such as synchronization of shared data, implementation of abstract data types, and scheduling of processes need to be addressed formally. In this paper, we seek to redress the situation by investigating in some detail the steps required to refine a high-level Unity solution to a specific problem to an efficient C implementation. In the process, we develop some theorems about program refinements. Though by no means complete, these theorems are adequate for the following problem that we consider from the domain of genetic sequence analysis.

We are given a number of finite strings over some finite alphabet (for DNA/RNA experiments, this alphabet is usually {a,c,g,t} or {a,c,g,u}). Some patterns over the alphabet are defined to be "critical". (For brevity, we omit the conditions under which a pattern is critical.) Two or more strings that share a critical pattern can be "pinned" or aligned together by that pattern. For example, consider the following two strings:

```
aaugacggguacacauaucaacuucacaggagcu
acuuacgcgcacaucucaacuucacagcagcu
```

If the pattern 'caacuucaca' is "critical", then it "pins" the two sequences. Such "pinning" can be used to establish overall correspondences, and then attempts are normally made to "align" the regions between pins [7]. The algorithm that we implement constructs the set of all "pins", given the positions of occurrences of critical substrings. We chose this particular problem because its specification is very simple and we are able to deal with implementation issues without getting lost in details.

We carry out our program development in Unity, a formalism developed by Chandy and Misra [1] for deriving programs and reasoning about them. We give a brief introduction to Unity in the next section. In Section 3, we present some theorems about program transformation that are used later. In Section 4, we discuss the formal specification of the problem and our solution strategy. In Section 5, we present the initial solution and a program refinement that deals with implementation of the abstract data type of sets. In Section 6, we discuss the partitioning of the statements over a set of processes and also specify and use an underlying scheduling mechanism to schedule these processes. In Section 7, we use an underlying mutual exclusion algorithm to address the problem of synchronizing shared resources. Section 8 includes concluding remarks. Finally, the appendix contains a C program that corresponds (albeit roughly) to the final Unity program.

2 The Unity Formalism

It is difficult to do justice to the entire scope and breadth of Unity in a few pages. What we present here has been much simplified; the reader may wish to consult [1] for the full details. We present the Unity syntax in Section 2.1 and the Unity logic in Section 2.2.

2.1 The Unity Syntax

A Unity program consists of four sections — a **declare** section that declares the variables used in the program, an **always** section that consists of a set of proper equations, an **initially** section that describes the initial values of the variables, and an **assign** section that consists of a non-empty set of assignment statements. The **declare** and the **initially** sections are not discussed any further, as all variable declarations are left implicit in our programs, and the **initially** section consists simply of a predicate on the program variables.

The **always** section is used to define certain program variables as functions of other variables. One of the main advantages of defining a variable this way is that decisions concerning how the variable is actually going to be implemented can be deferred to a later point. This eases program refinements as we worry about evaluations and implementations of variables one at a time.

The **assign** section consists of a non-empty set of assignment statements. An assignment statement consists of one or more assignment components separated by $\|$. An assignment component is either an enumerated assignment or a quantified assignment. An enumerated assignment has a variable list on the left, a corresponding expression list in the middle, and a boolean expression on the right (which by default is *true*):

$< \textit{variable-list} > := < \textit{expression-list} >$ **if** $< \textit{condition} >$.

A quantified assignment specifies a quantification and an assignment that is to be instantiated with the given quantification; a quantification names a set of bound variables and a boolean expression (the range) satisfied by the instances of the bound variables.

Examples: Examples of enumerated assignments are:

1. Exchange x, y, provided predicate b holds.
   $x, y := y, x$ **if** b
2. Add $A[i]$ into sum and increment i, provided i is less than N.
   $sum, i := sum + A[i], i + 1$ **if** $i < N$ □

Examples: Examples of quantified assignments are:

1. Assign all components of array $A[0..N]$ to 0.
   $\langle \| i : 0 \leq i \leq N :: A[i] := 0 \rangle$

2. Given arrays $A[0..N]$ and $B[0..N]$ of integers, assign the maximum of $A[i]$ and $B[i]$ to $A[i]$, for all $0 \leq i \leq N$.
   $\langle \| i : 0 \leq i \leq N :: A[i] := max(A[i], B[i]) \rangle$ □

An assignment component is executed by first evaluating all expressions and then assigning the values of the evaluated expressions to the appropriate variables, if the associated boolean expression is *true*; otherwise, the variables are left unchanged.

The set of assignment statements is written down either by enumerating every statement singly and using $[]$ as the set constructor, or by using a quantification of the form $\langle [] \ var : range : statement \rangle$. The symbol $[]$ is called the union operator.

A program execution starts from any state satisfying the initial conditions and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following fairness rule: Every statement is selected infinitely often [1].

Examples: The following program assigns the maximum of variables x and y to variable z. Its assign section consists of two assignment statements each of which has one assignment component consisting of an enumerated assignment.

Program *max*
  initially $z = 0$
  assign
    $z := x$ **if** $x > y$
  $[]$ $z := y$ **if** $x \leq y$
end

As another example consider the following program which sorts integer array $A[0..N]$, $N \geq 0$, in ascending order by swapping adjacent elements if they are out of order. Its assign section consists of N statements, one for every pair of adjacent positions.

Program *sort1*
  assign
    $\langle [] \ i : 0 \leq i < N :: A[i], A[i+1] := A[i+1], A[i]$ **if** $A[i] > A[i+1] \rangle$
end

As another example, consider the following program which is obtained from the previous program by combining all the N assignment statements into two assignment statements, one for even i and the other for odd i.

Program *sort2*
  assign
    $\langle \| i : even(i) :: A[i], A[i+1] := A[i+1], A[i]$ **if** $A[i] > A[i+1] \rangle$
  $[]$ $\langle \| i : odd(i) :: A[i], A[i+1] := A[i+1], A[i]$ **if** $A[i] > A[i+1] \rangle$

end

Observe that each assignment statement in the above program consists of one assignment component, which in turn consists of a single quantified assignment.

As a final example, consider the following program which is obtained by combining the two assignment statements of the previous program into a single assignment statement consisting of two assignment components.

Program *sort3*
assign
$$\langle \| i : even(i) :: A[i], A[i+1] := A[i+1], A[i] \quad \textbf{if} \quad A[i] > A[i+1] \rangle$$
$$\| \langle \| i : odd(i) :: A[i], A[i+1] := A[i+1], A[i] \quad \textbf{if} \quad A[i] > A[i+1] \rangle$$
end

□

Notation: The fixed point of a program, usually represented by FP, denotes the state of the program upon termination; it is obtained by replacing the assignment symbol := by the equality symbol = in every statement of the program and taking the conjunction over all such predicates. For example, the fixed point of program *max* is

$(x > y \Rightarrow z = x) \wedge (x \leq y \Rightarrow z = y)$. □

2.2 The Unity Logic

Program properties are expressed using four relations on predicates — *unless*, *invariant*, *ensures*, and *leads-to*. The first two are used for stating safety properties whereas the last two are used for stating progress properties.

2.2.1 Unless

For any two predicates p and q, the property p *unless* q holds in a program iff for all statements s in the program

$$\{p \wedge \neg q\}\ s\ \{p \vee q\}.$$

Informally, if p is true at some point in the computation, then either q never holds and p holds forever, or q holds eventually and p continues to hold until q holds.

Examples:

1. The value of x never decreases.
   $x = k$ *unless* $x > k$, or
   $x \geq k$ *unless false*

2. Philosopher u stays *hungry* until *eating*.
   hungry.u unless eating.u

3. In Program max, variable z retains its old value until it gets $max(x, y)$.
   $z = k$ *unless* $z = max(x, y)$ □

Notation: If the property p *unless false* holds in some program, then we say that the predicate p is *stable* in that program. Note that the fixed point of a program is always stable. □

2.2.2 Invariant

For any predicate p, the property *invariant* p holds in a program iff p holds initially and the program never falsifies p, i.e.,

$$\textit{initially } p \ \wedge \ p \textit{ unless } \textit{false}.$$

Examples:

1. Variable x is always positive.
   invariant $x \geq 0$

2. An eating philosopher u has all the required forks.
   invariant $eating.u \ \Rightarrow \ hasfork.u$ □

An invariant can be substituted for *true* and vice-versa in the context of a program. This referred to as the substitution axiom. Sometimes, for *invariant* p, we simply write p.

2.2.3 Ensures

For any two predicates, p and q, the property p *ensures* q holds in a program iff p *unless* q holds in the program and there exists a statement s in the program such that

$$\{p \wedge \neg q\} \ s \ \{q\}.$$

Thus, if p is true at some point in the computation then q holds eventually and p continues to hold until q holds. Statement s that establishes q is called the helpful statement.

2.2.4 Leads-to

The relation *leads-to* is denoted as $\mapsto$, and is defined to be the strongest relation satisfying the following three rules.

- p **ensures** $q \ \Rightarrow \ p \mapsto q$,
- $(p \mapsto q \ \wedge \ q \mapsto r) \ \Rightarrow \ p \mapsto r$, and
- For any set W,
  $(\forall m : m \in W : p.m \mapsto q) \ \Rightarrow \ ((\exists m : m \in W : p.m) \mapsto q)$.

The first two rules imply that $\mapsto$ includes the transitive closure of ensures and the third rule allows us to induct over sets. Given that $p \mapsto q$ in a program, we can assert that once p becomes *true*, eventually q becomes *true*. However, unlike p *ensures* q, we cannot assert

that p will remain *true* as long as q is not.
Examples:

1. A hungry philosopher u eventually *eats*.
   $hungry.u \mapsto eating.u$

2. If a message m is sent, then it is eventually received.
   $send.m \mapsto receive.m$

3. In Program *max*,
   $true \mapsto z = max(x, y)$ □

2.2.5 Program Composition by Union

Let F and G be programs with compatible **declare** sections (i.e., the declaration of the variables are compatible), compatible **always** sections (i.e., the two sets of equations are consistent), and compatible **initially** sections (i.e., the initial values of the variables are non-conflicting). Then, their composition is a new program denoted $F \,[\!]\, G$; every section of this program is obtained by a union of the corresponding sections of F and G. The following theorem follows from the definitions of **unless** and **ensures**.

- union theorem:

  p *unless* q in $F \,[\!]\, G \;\equiv\; p$ *unless* q in $F \;\wedge\; p$ *unless* q in G, and

  p *ensures* q in $F \,[\!]\, G \;\equiv\; (p$ *ensures* q in $F \;\wedge\; p$ *unless* q in $G) \vee$
  $(p$ *unless* q in $F \;\wedge\; p$ *ensures* q in $G)$

2.2.6 Program Composition by Superposition

Superposition is another mechanism to structure programs. Suppose we are given a program F and a statement r that does not assign to any of the variables of F. Then, the statement r can be superposed on program F in two ways – either it can be combined with a statement s of F to yield an augmented statement $s \| r$, or it can be added by itself to F, thus resulting in the composite program $F \,[\!]\, r$. In either case, all the properties of the original program are preserved. Moreover, the fixed point of the transformed program implies the fixed point of the original program. The above result is referred to as the superposition theorem.

3 Some Program Transformations

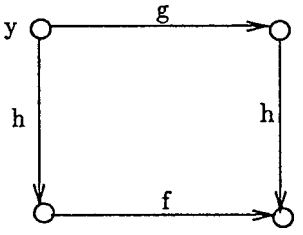
In this section we discuss three program transformations that preserve *unless* and *leads-to* properties (i.e., if the original program satisfies a *unless* or a *leads-to* property, then so does the transformed program), and preserve the fixed point (i.e., if the original program terminates, then so does the transformed program and moreover, the fixed point of

the transformed program implies the fixed point of the original program). We use these transformations to refine programs in Sections 4, 5, and 6. (For a formal proof of these transformations, see [8].)

3.1 Implementation of Abstract Data Types

Consider a program with a variable x such that x is initially a, and some statement in the program assigns $f(x)$ to x. Here x represents an abstract data type and f represents an operation on the abstract data type. For example, x may be a set and f may be the operation of adding an element to the set. We wish to examine conditions under which the abstract data type x can be implemented by a concrete data type y. Continuing with the example of sets, y may be an array of elements and we ask the question when can y implement x.

Suppose we define (in the always section) a function H (called the abstraction function) from the type of y to the type of x, replace $x = a$ in the initially section by $y = b$, and replace the assignment of $f(x)$ to x by the assignment of $g(y)$ to y. Then this transformation is correct if $a = H(b)$ and $H(g(y)) = f(H(y))$, for all y. The second condition can be restated by saying that the following diagram commutes.



3.2 Strengthening of Guards

Let F be a program and, $s :: A$ if Q and $t :: A$ if $Q \wedge R$, be two statements. Let f be a function with domain as the cartesian product of the types of the program variables and range as the natural numbers such that the value of the function never increases during computation, i.e.,

$f \leq k$ *unless* $false$.

Then $F \parallel t$ is a correct transformation of $F \parallel s$ if the following two conditions hold in F.

- $Q \mapsto R$, and
- $R \wedge f = k$ *unless* $\neg Q \vee f < k$.

It is straightforward to show that the strengthening of guard preserves safety properties (in this case *unless* properties); so, we concentrate on showing (rather informally) why the

above two conditions ensure that all *leads-to* properties are preserved. Because *leads-to* is defined inductively in terms of *ensures*, it is sufficient to show that if b *ensures* c is a property of the original program, then $b \mapsto c$ is a property of the transformed program.

Assume that statement s is the only helpful statement for the property b *ensures* c (otherwise, the proof follows from noting that the helpful statement is unchanged and all *unless* properties are preserved in the transformed program). Consider an execution of $F \parallel t$ in which the predicate $b \wedge \neg c$ becomes true at some point. Because the predicate Q must hold for an effective execution of statement s, eventually either statement t executes effectively (thus setting c to true), or the predicate $b \wedge \neg c \wedge Q$ becomes stable (becomes true and stays true). In the latter case, because of the two conditions described earlier, the predicate $b \wedge \neg c \wedge Q \wedge R$ also becomes stable. That means that statement t is continuously enabled and therefore, will eventually execute effectively, thus setting c to true. (A formal proof appears in [8]). Observe that if R *unless* $\neg Q$ holds in F, then we can satisfy the second condition by choosing f to be any constant function.

3.3 Transforming Quantification

Let s :: A if $\langle \exists i : i \in D :: p.i \rangle$ and t :: $\langle \parallel i : i \in D :: A$ if $p.i \rangle$ be two statements such that the domain D is finite and variable i does not occur free in A. The transformation in which s is replaced by t in the context of a program (assuming that the transformation is syntactically legal) is correct if the program maintains each $p.i$ as stable (i.e., satisfies the property $p.i$ *unless false*, for each i). (The reverse transformation in which statement t is replaced by statement s preserves all *unless*, *ensures*, and *leads-to* properties unconditionally).

4 Specification of the Problem

We are given sets $S.j$ (j ranging over some given domain) of pairs of natural numbers. Every set $S.j$ corresponds to a DNA string and the pair (i, k) is an element $S.j$ iff "identifier" i occurs at "position" k in the corresponding string. The number of identifiers in a particular string is expected to be much less than the size of the identifier space, i.e., each set $S.j$ is sparse in the first component.

Notation: In the rest of this paper, variable i ranges over the "identifiers", variable j ranges over the indices of the given sets, and variable k ranges over the "positions". □

A maximal matching, or a *pin*, for a given identifier i, is the maximal set of pairs (j, k) where the pair (i, k) belongs to set $S.j$. In other words,

$$pin.i = \{(j, k) : (i, k) \in S.j\}.$$

Set *pins* is defined to be a set of maximal matchings such that a maximal matching is in

the set iff it has at least two pairs, i.e.,

$$pins = \{(i, pin.i) : |pin.i| > 1\}. \quad (P)$$

We are required to write a program that satisfies the following two conditions.

- $true \mapsto FP$, $(A0)$
  i.e., the program eventually terminates, and
- $FP \Rightarrow P$, $(A1)$
  i.e., at the fixed point of the program, the property P holds.

We design our solution using the following strategy. We first develop a very simple program that meets the above two conditions. Then, we transform that program one step at a time. Sometimes, these transformations are dictated by efficiency concerns and at other times by the choice of target architecture. However, all of these transformations preserve the above two conditions $(A0)$ and $(A1)$, i.e., if the original program satisfies the two conditions then so does the transformed program. The correctness of the final program then follows from the correctness of the initial program.

The program transformations and the order in which we carry them out are as follows. The first transformation deals with our choice of set implementation. The next transformation deals with the computation of the set *pins*. After that we target our transformations towards the specific computer architecture in mind, which in this case is a shared memory MIMD system. In these transformations, we illustrate a general scheme to assign statements to processes. We also specify an underlying scheduling mechanism and use its properties to schedule the processes. Our final transformation addresses controlling access to shared variables.

Note: The initial solution and the sequence of transformations discussed here is by no means the only one. Because our aim is to illustrate an application of formal methods for parallel processing, we choose and explain one (hopefully, a simple one) of these alternatives. □

5 Initial Solution and Transformations

In this section we describe the initial solution and the first few transformations. The initial solution is developed in Section 5.1 and the transformations are discussed in the succeeding subsections.

5.1 Initial Solution

Observe that the set *pins* is defined in terms of the sets *pin.i*. Therefore, we first concentrate on computing the sets *pin.i* and delay decisions about the computation of the set *pins* by including property P in the always section. Let FP represent the fixed point of the

program that we are going to derive. We propose that $FP \equiv P$. Then the second condition of correctness, $(A1)$, follows immediately. Because of property P, the first condition of correctness can be restated as follows. For all i,

$$true \mapsto (pin.i = \{(j,k) : (i,k) \in S.j\}) ,$$

or, equivalently, for all i, j, k,

$$true \mapsto (i,k) \in S.j \equiv (j,k) \in pin.i.$$

We propose the following invariant for our program.

$(j,k) \in pin.i \Rightarrow (i,k) \in S.j$ This invariant is made to hold initially by initializing each $pin.i$ to the empty set, and maintained by adding the pair (j,k) to $pin.i$ only if $(i,k) \in S.j$. Thus, the proof obligation for the first correctness condition, $(A0)$, reduces to showing that, for all i, j, k,

$(i,k) \in S.j \mapsto (j,k) \in pin.i.$

The following program follows immediately from the above considerations.

Program *sol*0
always
$pins = \{(i, pin.i) : |pin.i| > 1\}$ $(E0)$
initially
$\langle [\!] \; i :: pin.i = \emptyset \rangle$
assign
$\langle [\!] \; i,j,k :: pin.i := pin.i \cup \{(j,k)\} \;\; \textbf{if} \;\; (i,k) \in S.j \rangle$
end

5.2 A Data Transformation

At this time we make a design choice about the concrete representation of sets $pin.i$. Led by the observation that the set of natural number pairs (i,k) is sparse in the first component, we use a hash table representation. The data items to be stored in the hash table are (i,j,k) where (i,k) is a member of $S.j$. We use a hash function h on the first argument i, and store the triple (i,j,k) at the $h(i)$th cell of the hash table.

Notation: Variable m ranges over the cells of the hash table. We denote the mth cell of the hash table by $bin.m$. □

Now, we use the theorem about implementation of abstract data types stated in Section 3.1 to transform program *sol*0. We implement set $pin.i$ as the set of all entries in the $h(i)$th cell of the hash table that have i as their first component. Drawing a correspondence with the statement of the theorem in Section 3.1, here x, the abstract data type, is the set $pin.i$ and y, the concrete data type, is the set $bin.(h.i)$. Our abstraction function H (for a given

i) is as follows.

$$H(bin.(h.i)) \;=\; \{(j,k) : (i,j,k) \in bin.(h.i)\} \qquad (E1)$$

In order to use the transformation correctly, we have to meet two proof obligations, the first on the initial conditions and the second on the assignment statements. Because $H(\phi) = \phi$, the first obligation is met by initializing $bin.(h.i)$ to the empty set. We discuss how the second proof obligation is met next.

For a given value of j and k, the function f that changes $pin.i$, our abstract data type, in program $sol0$ is given by:

$$f(pin.i) \;=\; pin.i \cup \{(j,k)\}$$

Accordingly, we define our function g, the function that changes the concrete data type, by:

$$g(bin.(h.i)) \;=\; bin.(h.i) \cup \{(i,j,k)\}$$

For a proof that the above definition of g satisfies the condition $H(g(bin.(h.i))) \;=\; f(H(bin.(h.i)))$, which is our final proof obligation, observe the following.

$H(g(bin.(h.i)))$
$=$ {definition of g}
$H(bin.(h.i) \cup \{(i,j,k)\})$
$=$ {definition of H and changing bound variables j,k to j',k'}
$\{(j',k') : (i,j',k') \in bin.(h.i)\} \cup \{(j,k)\}$
$=$ {definition of H and changing bound variables j,k to j',k'}
$H(bin.(h.i)) \cup \{(j,k)\}$
$=$ {definition of f}
$f(H(bin.(h.i)))$

The resulting program after these transformations is as follows.

Program $sol1$
always
$pins \;=\; \{(i, pin.i) : |pin.i| > 1\}$ $\qquad (E0)$
$\langle \,[\!]\; i :: pin.i \;=\; \{(j,k) : (i,j,k) \in bin.(h.i)\}\rangle$ $\qquad (E1)$
initially
$\langle \,[\!]\; m :: bin.m \;=\; \emptyset\rangle$
assign
$\langle \,[\!]\; i,j,k :: bin.(h.i) \;:=\; bin.(h.i) \cup \{(i,j,k)\}$ **if** $(i,k) \in S.j\rangle$
end

The above program satisfies correctness conditions $A0$ and $A1$ because program $sol0$ does so, and because our transformation preserves the fixed point.

5.3 Computing *pins*

In this section we make some decisions about the computation of *pins* which we had delayed so far (by defining it in the always section). Because every cell of the hash table stores a disjoint part of the set, we partition *pins* into m subsets, $pins.m$, one for each cell of the hash table, and then merge these partitions together to obtain *pins*. Sets $pins.m$ are defined by the following equation included in the always section:

$$pins.m \;=\; \{(i, pin.i) : h.i = m \;\wedge\; |pin.i| > 1\}. \qquad (E2)$$

In order to merge the sets $pins.m$, we add the following assignment statement (instantiated for each m) to the assign section.

$$pins \;:=\; pins \cup pins.m \;\; \textbf{if} \; ...$$

We discuss the missing predicate next.

Observe that the missing predicate should imply that the set $pins.m$ has obtained its final value. By virtue of equation $(E2)$, this means that all the cells of the hash table have obtained their final value, i.e.,

$$\langle \forall i, j, k : (i, k) \in S.j :: (i, j, k) \in bin.(h.i) \rangle.$$

In order to detect the above condition, we choose variables $c.i.j.k$, one for each i, j, k, and propose the following invariant.

$$c.i.j.k \;\equiv\; (i, j, k) \in bin.(h.i)$$

This invariant is made to hold initially by initializing variable $c.i.j.k$ to false and by setting it to true when the corresponding triple is added to the hash table. With all these changes, we have the following program.

Program *sol*2
always
$\langle \,[]\, i :: pin.i \;=\; \{(j, k) : (i, j, k) \in bin.(h.i)\}\rangle$ $(E1)$
$\langle \,[]\, m :: pins.m \;=\; \{(i, pin.i) : h.i = m \wedge |pin.i| > 1\}\rangle$ $(E2)$
initially
$pins \;=\; \emptyset$
$[]\, \langle \,[]\, m :: \;\; bin.m \;=\; \emptyset\rangle$
$[]\, \langle \,[]\, i, j, k :: \;\; c.i.j.k \;=\; false\rangle$
assign
$\langle \,[]\, i, j, k :: bin.(h.i), c.i.j.k \;\; := \;\; bin.(h.i) \cup \{(i, j, k)\}, \; true \;\; \textbf{if} \;\; (i, k) \in S.j \wedge \neg c.i.j.k\rangle$
$[]\, \langle \,[]\, m :: \;\; pins \;\; := \;\; pins \cup pins.m \;\; \textbf{if} \;\; \langle \forall i, j, k : (i, k) \in S.j :: c.i.j.k\rangle$
end

The above program terminates and its fixed point implies the fixed point of program *sol*1; therefore, it satisfies correctness conditions $A0$ and $A1$.

6 Assigning statements to processes

In this section we target our refinements towards the specific computer system in mind, which happens to be a shared memory MIMD system. We carry out the assignment of statements to processes by first identifying groups of statements that access common variables and then later partitioning these groups of statements over the set of processes. (This is reminiscent of identifying "schedulable units of work" [5].).

6.1 Partitioning the Statements

The assignment statements of program *sol*2 can be syntactically divided into two sets that are separated by a $[]$. For the first set of statements, we group together statements that access the same set $S.j$; we call this group of statements $A.j$. For the second set of statements, we group together statements that access the same cell of the hash table; we call this group of statements $B.m$. Thus, the total number of groups equals the number of sets $S.j$ plus the number of cells in the hash table. Formally, groups $A.j$ and $B.m$ are defined as follows:

$A.j = \langle [] \; i,k :: \; s.i.j.k \;\; \textbf{if} \;\; u.i.j.k \rangle$, and
$B.m = t.m \;\; \textbf{if} \;\; v.m$

where $s.i.j.k$, $t.m$, $u.i.j.k$, and $v.m$ are defined as follows.

$s.i.j.k :: \; bin.(h.i), c.i.j.k \;\; := \;\; bin.(h.i) \cup \{(i,j,k)\}, \; true$

$t.m :: \; pins \;\; := \;\; pins \cup pins.m$

$u.i.j.k :: \; (i,k) \in S.j \wedge \neg c.i.j.k$ and

$v.m :: \; \langle \forall i,j,k : (i,k) \in S.j :: c.i.j.k \rangle$.

With the above definitions the assign section of program *sol*2 can be abbreviated as follows:

$\langle [] \; j \; :: A.j \rangle$
$[] \; \langle [] \; m :: B.m \rangle$

6.2 Assigning Statement Groups to Processes

In the previous section, we identified groups of statements that access common variables. In order to partition the groups over the set of processes, we assume the existence of an underlying program *map* (which is similar to a scheduler) with the following properties.

Notation: Variables n, n' range over the processes. Predicate $p.j.n$ denotes that the group $A.j$ is assigned to process n. Similarly, predicate $q.m.n$ denotes that the set $B.m$ is assigned to process n. □

- $p.j.n \wedge p.j.n' \;\Rightarrow\; n = n'$, for all j, n, n',
  $q.m.n \wedge q.m.n' \;\Rightarrow\; n = n'$, for all m, n, n', and $\quad (C0)$

  i.e., a group is assigned to at most one process.

- $p.j.n$ *unless* $false$, for all j, n,
  $q.m.n$ *unless* $false$, for all m, n, and $\quad (C1)$

  i.e., the assignments, once made, are not changed.

- $true \mapsto \langle \exists n :: p.j.n \rangle$, for all j, n,
  $true \mapsto \langle \exists n :: q.m.n \rangle$, for all m, n, and $\quad (C2)$

  i.e., every group of statements is eventually assigned.

(C3) Program *map* terminates and does not modify any variables of program *sol*2. $(C3)$

Note: There are various ways to implement the program *map*. One alternative is to partition the sets and the cells of the hash table statically. Though simple, this solution may not perform well. Another solution is to have two variables that indicate, respectively, the next set and the next cell of the hash table to be assigned. Then the processes access these variables (using mutual exclusion), obtain an "unit of work" to be done, and then update these variables. However, the implementation of program *map* should not be our concern at this point.

It follows from condition $(C3)$ and the superposition theorem that the the composite program *sol*2 [] *map* also satisfies correctness conditions $(A0)$ and $(A1)$.

In the next refinement, we strengthen the guard of every statement by applying the transformation suggested in Section 3.2. The statements in group $A.j$ are strengthened by the predicate $\langle \exists n :: p.j.n \rangle$ (predicates Q and R for this group are $true$ and $\langle \exists n :: p.j.n \rangle$, respectively), and the statements in group $B.m$ are strengthened by the predicate $\langle \exists n :: q.m.n \rangle$ (predicates Q and R for this group are $true$ and $\langle \exists n :: q.m.n \rangle$, respectively). The first correctness condition of the transformations (i.e., $true \mapsto \langle \exists n :: p.j.n \rangle$ and $true \mapsto \langle \exists n :: q.m.n \rangle$) follows from condition $(C2)$. For the second correctness condition of the transformation, we choose the non-increasing function to be any constant function, and our proof obligations ($\langle \exists n :: p.j.n \rangle$ *unless* $false$ and ($\langle \exists n :: q.m.n \rangle$ *unless* $false$) follow from applying disjunction on condition $(C1)$.

After applying the transformations, the set of assignment statements can be now rewritten as follows:

$\langle \; [] \; j :: A.j \;\; \textbf{if} \;\; \langle \exists n :: p.j.n \rangle \rangle$
$[] \; \langle \; [] \; m :: B.m \;\; \textbf{if} \;\; \langle \exists n :: q.m.n \rangle \rangle$

Next, we apply the transformation suggested by in Section 3.3. The correctness of this transformation follows from condition $(C1)$. We obtain the following set of statements.

$\langle \,[\!]\, n :: \langle \,[\!]\, j :: A.j \;\; \textbf{if} \;\; p.j.n \rangle \rangle$
$[\!]\, \langle \,[\!]\, n :: \langle \,[\!]\, m :: B.m \;\; \textbf{if} \;\; q.m.n \rangle \rangle$

Reordering the statements we have the following program.

Program *sol3*
always
$\langle \,[\!]\, i :: pin.i \;=\; \{(j,k) : (i,j,k) \in bin.(h.i)\} \rangle$ $(E1)$
$\langle \,[\!]\, m :: pins.m \;=\; \{(i, pin.i) : h.i = m \wedge |pin.i| > 1\} \rangle$ $(E2)$
initially
$pins \;=\; \emptyset$
$\langle \,[\!]\, m :: \;\; bin.m \;=\; \emptyset \rangle$
$[\!]\, \langle \,[\!]\, i,j,k :: \;\; c.i.j.k \;=\; false \rangle$
assign
$\langle \,[\!]\, n :: \langle \,[\!]\, j :: A.j \;\; \textbf{if} \;\; p.j.n \rangle$
$[\!]\, \langle \,[\!]\, m :: B.m \;\; \textbf{if} \;\; q.m.n \rangle$
$\rangle$
$[\!]\, map$
end

The code for a process is then obtained by instantiating the body of the outermost quantification (in the assign section); thus, the code for process n is as follows.

$\langle \,[\!]\, j :: A.j \;\; \textbf{if} \;\; p.j.n \rangle$
$[\!]\, \langle \,[\!]\, m :: B.m \;\; \textbf{if} \;\; q.m.n \rangle$

7 Synchronizing Access to Shared Resources

In this section we examine the program in order to identify the shared resources for which the processes may contend, and then use a mutual exclusion scheme to synchronize access to these resources. We observe the following shared resources in program *sol3* – cells $bin.m$, for each m, and set *pins* and use a mutual exclusion scheme for each of these shared resources.

In order to carry out the mutual exclusion, we assume a program *mutex* and a set of variables $mode.n.x$ for every shared resource x and every process n. Variable $mode.n.x$ stores the state of process n with respect to the resource. It can take one of three distinct values – t, h, or e. These values are interpreted as follows: if $mode.n.x = t$ then process n is not interested in resource x, if $mode.n.x = h$ then process n is waiting to access resource x, and if $mode.n.x = e$ then process n is accessing resource x. The sequence of state transitions if from t to h to e to t. The specification of program *mutex* is as follows. (The above specification is based on the specification of the mutual exclusion problem in [1].)

- The only transitions of $mode.n.x$ that are made by *mutex* are from h to e, i.e., $mode.n.x = t$ *unless* $false$, and $mode.n.x = e$ *unless* $false$, for all n, x.
- Two different processes do not access a resource at the same time, i.e., $mode.n.x = e \ \wedge \ mode.n'.x = e \ \Rightarrow \ n = n'$, for all n, n', x.
- If every process that has a resource eventually relinquishes it then every process that is waiting for a resource eventually gets it, i.e, if $mode.n.x = e \ \mapsto \ mode.n.x = t$, for all n, x, then $mode.n.x = h \ \mapsto \ mode.n.x = e$, for all n, x.
- It terminates and does not modify any variables of program *sol*3.

Let s be any assignment statement of *sol*3 that accesses a shared resource x and Q be the guard for the statement. Observe that Q is local to the process (i.e., does not mention any shared variables) and is stable over the rest of the program. We carry out the following transformation for each such statement.

- augment the assignment statement $mode.n.x \ := \ t \ \textbf{if} \ Q \ \wedge \ mode.n.x = e$ to s, and add the statement $mode.n.x \ := \ h \ \textbf{if} \ Q \ \wedge \ mode.n.x = t$
- strengthen the guard of s by adding the predicate $mode.n.x = e$.

Observe that the above transformation modifies variable $mode.n.x$ in the correct sequence, i.e., from t to h and from e to t. Moreover, the transition from e to t occurs eventually (i.e., every process eventually gives up the shared resource) because the predicate Q is stable over the rest of the program.

For a proof of correctness of the above transformations, observe the following. The first step preserves correctness conditions $(A0)$ and $(A1)$ because it is a legal superposition and the superposed program terminates. The second step is justified (i.e., it preserves all *unless* and $leads - to$ properties) by the theorem on strengthening of guards in Section 3.2. In order to apply the theorem, we have to show that two conditions are satisfied — first, we have to show that $Q \ \mapsto \ mode.n.x = e$ in the rest of the program, and next, we have to show that there exists a non-increasing function f of the program variables such that

$$mode.n.x = e \ \wedge \ f = k \ \ unless \ \ \neg Q \ \vee \ f < k$$

in the rest of the program. This is shown next. (Here R, the predicate which is added to the guard of the statement is $mode.n.x = e$.)

For the proof of the first condition, observe that from step 1 of the transformation and because Q is stable, $Q \ \wedge \ mode.n.x = t \ \mapsto \ Q \ \wedge \ mode.n.x = h$. Similarly, from the property of program *mutex*, $Q \ \wedge \ mode.n.x = h \ \mapsto \ Q \wedge \ mode.n.x = e$. From these two properties, we have that $Q \ \wedge \ mode.n.x \neq e \ \mapsto \ Q \wedge \ mode.n.x = e$. Consequently, $Q \ \mapsto \ Q \wedge \ mode.n.x = e$.

For the proof of the second condition of the theorem, consider the two types of statements that access shared resources separately. For the first type of statement which accesses the shared resource $bin.(h.i)$, $Q \equiv u.i.j.k \wedge p.j.n$. (Predicate $u.i.j.k$ was defined earlier in Section 6.1.) Define the function f which maps a state of the program to the natural numbers as follows: given the sets $S.j$ and the hash table, it returns the number of elements in the sets that have not been added to the hash table. Thus, the function f is bounded from below by 0 and non-increasing for our program. Observe that every transition of $mode.n.(bin.(h.i))$ from e to t decreases the function f. Therefore,

$$mode.n.(bin.(h.i)) = e \ \wedge \ f = k \ \ unless \ \ f < k$$

holds for the rest of the program. The required proof then follows from weakening of the the right hand side of the above property.

Next, consider the other type of statements that accesses the shared resource $pins$. Here, $Q \equiv v.m \wedge q.m.n$. Because statement s is the only statement that modifies the variable $mode.n.pins$ when it equals e,

$$mode.n.pins = e \ \ unless \ \ false$$

in the rest of the program. Thus, as observed in Section 3.2, we can define f to be any constant function and our proof obligation follows once again from weakening of the the right hand side of the above property.

This completes the proof of correctness of the above transformation and the resulting program is as follows.

Program *sol*4
always
$\langle [] \ i :: pin.i \ = \ \{(j,k) : (i,j,k) \in bin.(h.i)\}\rangle$ $(E1)$
$\langle [] \ m :: pins.m \ = \ \{(i, pin.i) : h.i = m \wedge |pin.i| > 1\}\rangle$ $(E2)$
initially
$pins \ = \ \emptyset$
$\langle [] \ m :: \ bin.m \ = \ \emptyset\rangle$
$[] \ \langle [] \ i,j,k :: \ c.i.j.k \ = \ false\rangle$
assign
$\langle [] \ n :: \langle [] \ j :: A.j$ **if** $p.j.n\rangle$
$[] \ \langle [] \ m :: B.m$ **if** $q.m.n\rangle$
$\rangle$
$[] \ map$
$[] \ mutex$
end

where sets of statements $A.j$ and $B.m$ are as follows.

$$A.j = \langle [] \ i,k :: \ mode.n.(bin.(h.i)) \ \ := \ \ h \ \textbf{if} \ u.i.j.k \wedge mode.n.(bin.(h.i)) = t$$

$$\| \; s.i.j.k \qquad \textbf{if } u.i.j.k \wedge mode.n.(bin.(h.i)) = e$$

$\rangle$

and

$$B.m = \; mode.n.pins \; := \; h \; \textbf{if } v.m \wedge mode.n.pins = t$$
$$\| \; t.m \qquad \textbf{if } v.m \wedge mode.n.pins = e$$

where $s.i.j.k$, $t.m$, $u.i.j.k$, and $v.m$ are defined as follows.

$s.i.j.k :: bin.(h.i), c.i.j.k, mode.n.(bin.(h.i)) := bin.(h.i) \cup \{(i,j,k)\}, true, t$

$t.m :: pins.mode.n.pins := pins \cup pins.m, \; t$

$u.i.j.k :: (i,k) \in S.j \wedge \neg c.i.j.k$ and

$v.m :: \langle \forall i,j,k : (i,k) \in S.j :: c.i.j.k \rangle$.

This concludes our program development.

8 Discussion

We hope to have shown by the preceding exercise that it is possible to carry out a lot of the implementation of a parallel program in the formal domain without losing any of the efficiency. Usually, during the derivation process a number of important design decisions that affect the simplicity of the final program (and therefore, the success of the formal method) are made. For example, we first decided to implement the sets by a hash table, then assigned statements to processes, and finally addressed the question of synchronization. Carrying out the derivation in a different order leads to a complicated solution and all the advantages of a formal derivation are lost. In general, the implementation of abstract data types should be addressed first, assignment of statements onto processes should be addressed next, and synchronization issues (such as locking and level of atomicity) should be addressed last.

The point where one stops the formal derivation process is also an important design decision. When all the important design decisions about the target architecture have been made, there is not much use continuing further. For example, we stopped our formal derivation when we had made our choices about implementation of sets, processes, and locking. We did not indicate explicitly how the sets $pins.m$ are computed as it was not an important design issue. The final step in obtaining a "running" program is to translate the final derived program to a "real" programming language supported on the target architecture. Because many of the important design decisions have already been made, this final step should not be difficult. In the appendix, we show a C program that was obtained by translating program $sol4$.

Our intent in presenting the C program is not to argue that it could be eventually automatically compiled from the Unity solution. Rather, we believe that the execution sequences of the C program are a subset of the execution sequences of the Unity program, and there-

fore, our proof of correctness for the Unity program is also valid for the C implementation.

Perhaps the biggest advantage of a formal program development is that refinements for different architectures share a lot of the initial development. For example, if we are to develop a solution to the same problem for process networks, our derivation is a lot similar; only now instead of locking shared resources, we define a communication scheme. Our partitioning of statements into groups is similar; we define groups $A.j$ as before, we merge all the groups $B.m$ into a single group B, and define a new group of statements $C.m$, one for every cell of the hash table, that computes set *pins.m*. (This set was earlier defined in the always section.) Each of these groups is assigned to a different process.

The process assigned to group $A.j$ takes the set $S.j$ as input and adds the elements of this set to the appropriate cell of the hash table by sending messages to processes in charge of the cell, i.e., the group C processes. For example, if the pair (i, k) is an element of the set $S.j$, then the process $A.j$ will send a message (i, j, k) to the process assigned to group $C.(h.i)$. The process assigned to group $C.m$ takes the messages sent by the group A processes as input and outputs the set *pins.m* to the group B process. The process assigned to group B takes all these sets *pins.m* as input and produces the set *pins* as the final output. A process network for 3 sets and 2 cells is illustrated below.

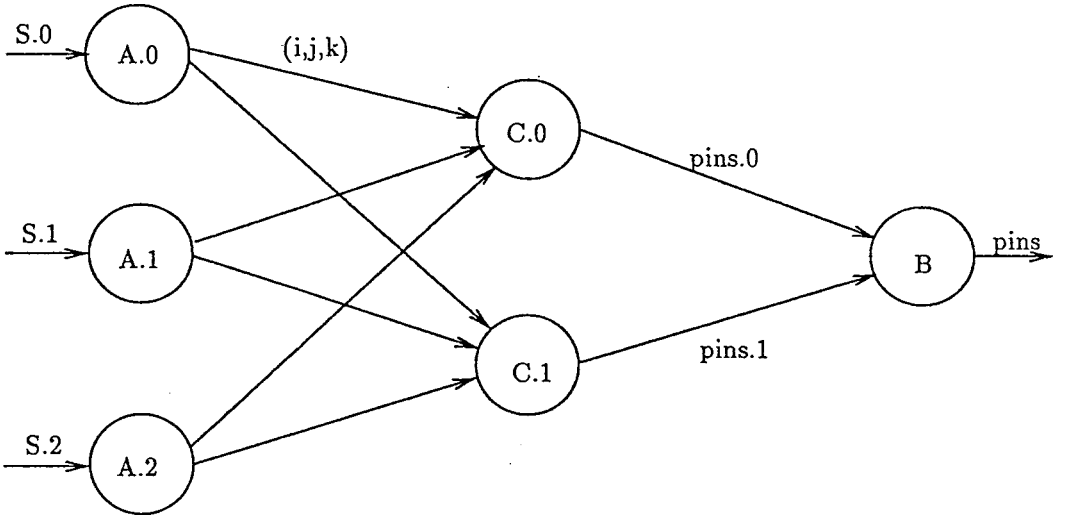


Figure: A process network for 3 sets and 2 bins

References

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

[2] Chandy, K. M., and J. Misra, An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," ACM Transactions on Programming Languages and Systems, 8:3, July 1986, pp. 326 – 343.

[3] Dijkstra, E. W., A Discipline of Programming, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

[4] Gries, D., The Science of Programming, New York: Springer-Verlag, 1981.

[5] Browne, J.C., M. Azam, and S. Sobek, "CODE: A Unified Approach to Parallel Programming," IEEE Software, July 1989, pp. 10 – 17.

[6] Lynch, N., and M. J. Fischer, "A Technique for Decomposing Algorithms which Use a Single Shared Variable," Journal of Computer and System Sciences, 27:3, December 1983, pp. 350 – 377.

[7] Butler, R., T. Butler, I. Foster, N. Karonis, R. Olson, N. Pflunger, M. Price, S. Tuecke, "Generating Alignments of Genetic Sequences," Technical Report ANL/MCS-TM-132, Mathematics and Computer Science Division, Argonne National Laboratories, June 1989.

[8] Singh, A. K., "On Strengthening the Guard," Notes on Unity – 7, June 1989.

Appendix: An Equivalent C Program

We implement program *mutex* by a locking scheme that is provided by C routines. For this purpose, we declare a lock variable with each shared object and define a few macros that ensure the correct transition of the lock variables. A t to h transition is implemented by a requesting a lock, an h to e transition is ensured by the underlying locking mechanism, and an e to h transition is implemented by releasing a lock. The complete set of macros is as follows. (For brevity, we skip the definition of the macros.)

- Macro DEC_MUTEX declares a mutual exclusion variable.
- Macro $MODE\_INIT$ initializes a mutual exclusion variable to the state t.
- Macro $MODE\_T\_TO\_H$ defines the transition of a mutual exclusion variable from t to h.
- Macro $MODE\_E\_TO\_T$ defines the transition of a mutual exclusion variable from e to t.

We implement program *map* by two macros $MAP\_GET\_SET(j)$ and $MAP\_GET\_BIN(m)$ that define the next set to be loaded in to the hash table and the next cell of the hash table to be added to the set *pins*, respectively. Thus, macro $MAP\_GET\_SET(j)$ implements the p predicates and macro $MAP\_GET\_BIN(m)$ implements the q predicates. The complete set of macros is as follows.

- Macro DEC_MAP declares the variables required to implement program *map*.
- Macro MAP_INIT initializes the variables declared by the above macro.
- Macro $MAP\_GET\_SET(j)$ implements the p predicates and sets j to the next set to be loaded into the hash table; j is set to -1 if all the sets have been allocated.
- Macro $MAP\_GET\_BIN(m)$ implements the q predicates and sets m to the next cell of the hash table to be added to the set *pins*; m is set to -1 if all the cells have been allocated.

Recollect from program *sol*4 that we use boolean variables $c.i.j.k$ to indicate that the pair (i, k) from set $S.j$ has been loaded into the hash table. Because it is difficult to implement all these variables (the cartesian product of the three domains may be quite large), we make a design decision. Instead of implementing variable $c.i.j.k$, we implement a single variable fs that counts the number of sets that have been loaded into the hash table. Then, if fs equals the total number of sets (ts), then all the sets have been loaded. Thus, the predicate $fs = ts$ detects the predicate $(\forall i, j, k : (i, k) \in S.j : c.i.j.k)$ [1]. This means that predicate $v.m$ in program *sol*4 can be replaced by the predicate $fs = ts$. Also, because we implement

the sets as a linked list that is processed sequentially, we no longer need the predicate $c.i.j.k$ as a part of the predicate $u.i.j.k$. Since, predicates $v.m$ and $u.i.j.k$ are the only places that we use the definition of boolean variable $c.i.j.k$, our design decision is appropriate.

Because now we decide to display the final output, we need to detect the fixed point of the Unity program. For this purpose, we use another variable fb that counts the number of cells of the hash table that have been added to the *pins*. Thus, the fixed point is reached when fb equals the number of cells in the hash table, NC.

We use the following macros for implementing variables fs and fb. (As before, we skip the definition of the macros for brevity.)

- Macro DEC_SYNCH declares the variables used.
- Macro $SYNCH\_INIT$ initializes the variables declared by the above macro.
- Macro INC_S increments the number of loaded sets by one.
- Macro $COMP\_S$ tests if all the sets have been loaded into the hash table.
- Macro INC_C increments the number of cells added to set *pins* by one.
- Macro $COMP\_C$ tests if all the cells have been added to set *pins*.

This completes the discussion of the macros used in the C implementation. Next, we discuss the data structures that are used.

We use the following constants in our C program.

NP:	The number of processes (actual UNIX processes).
NC:	The number of cells in the hash table.
MS:	The maximum number of sets in the input.
MR:	The maximum number of pairs in any single set.

Data structure *pin_set* is used to store the natural number pairs corresponding to a particular input set. It is a record consisting of an "id" for the set and a set of 2-tuples. The set of 2-tuples is implemented as a singly-linked list each node of which of the type *pin_entry*.

```
struct pin_set {
        struct pin_set *link;
        int i;
        struct pin_entry {
                struct pin_entry *link;
                int j;
                int k;
        } *pin_elements;
}
```

The following are the global variables used in our program: Array S stores the given sets of natural number pairs, array *tnum* stores the total number of pairs in each set, variable *ts* stores the total number of sets, array *bin* stores the hash table entries, and variable *pins* stores the final result. Macro DEC_MUTEX declares the mutual exclusion variables associated with the shared data structures, which (as discussed in Section 7) consist of set *pins* and every cell of the hash table. Macros DEC_MAP and DEC_SYNCH declare the variables needed for program *map* and for synchronization of the computation phases, respectively.

```
struct global {
        struct two_tuple {
                int i;
                int k;
        } S[MS][MR];
        int tnum[MS];
        int ts;
        struct pin_set *bin[NC];
        struct pin_set *pins;
        DEC_MUTEX(pins_lk);
        DEC_MUTEX(bin_lk[NC]);
        DEC_MAP;
        DEC_SYNCH;
} *glob;
```

This completes the discussion on the data structures used by the C implementation. Next, we discuss the subroutines that are used in the implementation.

The *main* program of the implementation first allocates the global memory, then reads the input sets and computes the required result, and finally prints the output.

```
main()
{
        glob = (struct global *) g_malloc(sizeof(struct global));
        read_input();
        compute_pins();
        present_output();
}
```

Subroutine *compute_pins* corresponds to the final Unity program *sol*4. It first calls the subroutine *init_section* to initialize all the variables (this corresponds to the **initially** section), and then forks the required number of *slave* processes (subroutine *slave* is defined later) to execute the **assign** section of the Unity program. Subroutine *init_local_env* sets up the environment for dynamic memory allocation.

```
compute_pins()
{
int x;
        init_section();
        for (x = 1; x < NP; x++) {
                create(slave);
        }
        init_local_env();
        execute_assign_section();
}
```

Subroutine *slave* is as follows.

```
slave()
{
        init_local_env();
        execute_assign_section();
}
```

Subroutine *init_section* initializes the variables and corresponds to the **initially** section of the Unity program.

```
init_section()
{
int x;
        glob-> pins = NULL;
        MODE_INIT(glob-> pins_lk);
        MAP_INIT;
        for (x = 0; x < NC; x++)
        {
                glob-> bin[x] = NULL;
                MODE_INIT(glob-> bin_lk[x]);
        }
}
```

The subroutine *execute_assign_section* corresponds to the set of assignment statements that is executed by each process. It corresponds to the following set of statements from the Unity program.

$\langle \, [\!] \; j :: A.j \;\; \textbf{if} \;\; p.j.n \rangle$
$[\!] \, \langle \, [\!] \; m :: B.m \;\; \textbf{if} \;\; q.m.n \rangle$

Subroutine *execute_assign_section* first calls *compute_A* to execute the set of assignment statements corresponding to the following set of Unity statements.

$\langle \, [\!] \; j :: A.j \;\; \textbf{if} \;\; p.j.n \rangle$

Next, it waits until all the sets have been loaded into the hash table (as pointed out earlier, macro $COMP\_S$ implements the predicate $v.m$ of the Unity program), and then calls *compute_B* to execute the set of assignment statements corresponding to the following set of Unity statements.

$[\!] \, \langle \, [\!] \; m :: B.m \;\; \textbf{if} \;\; q.m.n \rangle$

Finally, it calls the subroutine $COMP\_C$ in order to wait until a fixed point has been reached.

```
execute_assign_section()
{
        compute_A();
        COMP_S;
        compute_B();
        COMP_C;
}
```

Subroutines *compute_A* and *compute_B* are defined as follows.

```
compute_A()
{
int j;
        MAP_GET_SET(j);
        while (j!= -1)
        {
                proc_set_A(j);
                INC_S;
                MAP_GET_SET(j);
        }
}

compute_B()
{
int m;
struct pin_ *pins_m;
struct pin_set *proc_bin_E2();
        MAP_GET_BIN(m);
        while (m!= -1)
        {
                pins_m = proc_bin_E2(m);
```

```
                proc_bin_B(pins_m);
                INC_C;
                MAP_GET_BIN(m);
        }
}
```

Subroutine *proc_set_A* corresponds to the assignment statement $A.j$ of the Unity program, and is implemented as follows. Subroutine h implements the hashing function.

```
proc_set_A(j)
int j;
{
int x, h_i;
struct pin_set *p;
struct pin_entry *pe;
        for (x = 0; x < glob-> tnum[j]; x++)
        {
                h_i = h(glob-> S[j][x].i);
                MODE_T_TO_H(glob-> bin_lk[h_i])
                for (p = glob-> bin[h_i];
                        p&&(p-> i != glob-> S[j][x].i);
                        p = p-> link);
                if (!p)
                {
                        p = (struct pin_set*) u_malloc(sizeof(struct pin_set));
                        p-> link = glob-> bin[h_i];
                        glob-> bin[h_i] = p;
                        p-> i = glob-> S[j][x].i;
                        p-> pin_elements = NULL;
                }
                pe = (struct pin_entry *) u_malloc(sizeof(struct pin_entry));
                pe-> link = p-> pin_elements;
                pe-> j = j;
                pe-> k = glob-> S[j][x].k;
                p-> pin_elements = pe;
                MODE_E_TO_T(glob-> bin_lk[h_i]);
        }
}
```

Subroutine *proc_bin_B* corresponds to the assignment statement $B.m$ of the Unity program, and is implemented as follows.

```
proc_bin_B(pins_m)
struct pin_set * pins_m;
{
struct pin_set  * *p1;
        if (pins_m)
        {
                for (p1 = &pins_m; *p1; p1 = &((*p1)-> link));
                MODE_T_TO_H(glob-> pins_lk);
                *p1 = glob-> pins;
                glob-> pins = pins_m;
                MODE_E_TO_T(glob-> pins_lk);
        }
}
```

The following subroutine, *proc_bin_E2*, implements equation $E2$ in the **always** section of the Unity program.

```
struct pin_set  * proc_bin_E2(m)
int m;
{
struct pin_set  * pins_m, *p2, *p3;
        pins_m = NULL;
        p2 = glob-> bin[m];
        while (p2)
        {
                if (p2-> pin_elements-> link)
                {
                        p3 = p2-> link;
                        p2-> link = pins_m;
                        pins_m = p2;
                        p2 = p3;
                }
                else
                        p2 = p2-> link;
        }
        return(pins_m);
}
```