A Collection of Declarative Ada Example Programs

John Thornley
Computer Science Department
California Institute of Technology
Pasadena, California 91125, USA
john-t@cs.caltech.edu

April 24, 1993
Contents

1 Introduction 1

2 Lists, Trees, and Recursion 2
  2.1 Sorting Problem Specification 2
  2.2 Insertion Sort 3
    2.2.1 The Program 3
    2.2.2 Discussion 4
  2.3 Mergesort 4
    2.3.1 The Program 4
    2.3.2 Discussion 5
  2.4 Quicksort 6
    2.4.1 The Program 6
    2.4.2 Discussion 7
  2.5 Treesort 8
    2.5.1 The Program 8
    2.5.2 Discussion 9
  2.6 Heapsort 11
    2.6.1 The Program 11
    2.6.2 Discussion 12
  2.7 A Test Program 14
  2.8 Test Output 16

3 Arrays and Loops 17
  3.1 The All-Points Shortest-Path Problem 18
    3.1.1 Problem Specification 18
    3.1.2 The Program 19
    3.1.3 Discussion 19
    3.1.4 A Test Program 20
    3.1.5 Test Output 23
  3.2 Dirichlet’s Problem 24
    3.2.1 Problem Specification 24
    3.2.2 The Program 26
    3.2.3 Discussion 26
    3.2.4 A Test Program 27
    3.2.5 Test Output 29
1 Introduction

This report presents and discusses a collection of programs written in the Declarative Ada programming language [8]. It is intended as supplementary reading to [7] and [8].

The programs are grouped as follows:

1. Five sorting programs that use lists, trees, and recursion.
2. Two programs that use arrays and parallel loops.
3. Two programs that involve networks of processes that communicate using streams.
4. A larger program, dining philosophers, that includes nondeterminism.

Although related programs are grouped together, this report is not intended to be a systematic presentation of Declarative Ada features or programming paradigms. Nor is it intended to be a tutorial. The goals and methods of parallel programming with Declarative Ada are described in [7], which also discusses some of the examples in this report. Additionally, [3] and [4] describe parallel programming techniques with single-assignment variables.

In some cases, the programs could be more elegantly written if Declarative Ada was based on a larger subset of Ada [1]. For instance, enumeration types, unconstrained array types, variant record types, case statements, packages, and generic units would be useful. However, the overall structure of the programs and the methods of expressing parallelism would remain the same.

All of the programs presented are complete and have been compiled and executed using an implementation of Declarative Ada developed by the author [6]. A test program and actual test output is given for each program. Correct execution with a particular set of test data is no guarantee of a correct program. However, in practice methodical testing is an important part of program development.

The programs presented do not constitute a test suite for an implementation of Declarative Ada. An actual test suite is discussed in [6].
2 Lists, Trees, and Recursion

Dynamically allocated data structures such as lists and trees provide a means of communication and synchronization between parallel processes. The production and consumption of dynamic data structures can be executed in parallel. Synchronization occurs automatically: consumer processes suspend when they require an element that is undefined, and resume execution after the element is defined by the producer process. For instance, one process (the consumer) can be searching a list as it is being created by another process (the producer). Parallelism of this kind is often called pipeline parallelism.

Recursion is the method of creating and traversing dynamically allocated data structures.

In this section we present five different list sorting programs: insertion sort, mergesort, quicksort, treesort, and heapsort. The first three use only lists. The latter two use trees as intermediate data structures.

It is notable that, with the addition of appropriate packaging, each of the programs is a correct sequential Ada program to sort a list. The results of the programs are entirely independent of whether the execution is parallel or sequential.

2.1 Sorting Problem Specification

A function is required that satisfies the following specification:
type node;
type list is access node;
type node is record
   head : integer;
   tail : list;
end record;

function sort(unsorted : list) return list;
   -- Input Condition:
   -- unsorted is finite and acyclic.
   -- Output Condition:
   -- sort(unsorted) is a permutation of unsorted, and
   -- sort(unsorted) is in ascending order.

2.2 Insertion Sort

2.2.1 The Program

type node;
type list is access node;
type node is record
   head : integer;
   tail : list;
end record;

function insert(item : integer; items : list) return list is
begin
   if items = null then
      return new node'(item, null);
   else
      if item <= items.head then
         return new node'(item, items);
      else
         return new node'(items.head, insert(item, items.tail));
      end if;
   end if;
end insert;
function sort(unsorted : list) return list is
begin
  if unsorted = null then
    return null;
  else
    return insert(unsorted.head, sort(unsorted.tail));
  end if;
end sort;

2.2.2 Discussion

The insertion sort program demonstrates parallel execution derived from functional composition.

If the unsorted list is the empty list, the sort function returns the empty list (which is trivially in ascending order). Otherwise, the sort function returns the head of the unsorted list inserted into its correct position in the recursively sorted tail of the unsorted list. The recursion terminates (if the unsorted list is finite and acyclic), because each recursive call sorts a list with one fewer elements.

At each level of a sort call, the insert and recursive sort processes are executed in parallel. Therefore, the complete executing program consists of a pipeline of parallel sort and insert processes.

2.3 Mergesort

2.3.1 The Program

type node;
type list is access node;
type node is record
  head : integer;
  tail : list;
end record;
procedure split(items : in list; left, right : out list) is
    left.left, right.right : list;
begin
    if items = null or else items.tail = null then
        left := items;
        right := null;
    else
        split(in items.tail.tail, out left.left, out right.right);
        left := new node'(items.head, left.left);
        right := new node'(items.tail.head, right.right);
    end if:
end split;

function merge(left, right : list) return list is
begin
    if left = null then
        return right;
    elsif right = null then
        return left;
    elsif left.head <= right.head then
        return new node'(left.head, merge(left.tail, right));
    else
        return new node'(right.head, merge(left, right.tail));
    end if:
end merge;

function sort(unsorted : list) return list is
    left, right : list;
begin
    if unsorted = null or else unsorted.tail = null then
        return unsorted;
    else
        split(in unsorted, out left, out right);
        return merge(sort(left), sort(right));
    end if:
end sort;

2.3.2 Discussion

The mergesort program demonstrates parallel execution derived from both functional composition and parallel composition of statements. Use of functional composition alone can be unwieldy or inefficient when functions have multiple outputs, function calls are nested deeply, or the same function call
occurs in many places.

If the unsorted list is the empty list or a list with one element, the sort function returns the unsorted list (which is trivially in ascending order). Otherwise, the unsorted list is split into left and right sublists that together form a permutation of the unsorted list and differ in length by at most one element. The sort function returns the ordered merge of the recursively sorted left and right sublists.

Since the unsorted list has at least two elements, each sublist has at least one element and (if the unsorted list is finite and acyclic) has at least one fewer elements than the unsorted list. Therefore, the recursion terminates (if the unsorted list is finite and acyclic), because each recursive call sorts a list with fewer elements.

At each level of a sort call, the split, merge, and the two recursive sort processes are executed in parallel. In particular, the two recursive sort processes are independent of each other and are executed in parallel without interaction. Therefore, the complete executing program consists of a network of parallel sort, split, and merge processes.

2.4 Quicksort

2.4.1 The Program

```plaintext
type node;
type list is access node;
type node is record
    head : integer;
    tail : list;
end record;
```
procedure partition(pivot : in integer; items : in list; left, right : out list is
left_tail, right_tail : list;
begin
if items = null then
left := null;
right := null;
else
if items.head <= pivot then
partition(in pivot, in items.tail, out left_tail, out right);
left := new node'(items.head, left_tail);
else
partition(in pivot, in items.tail, out left, out right_tail);
right := new node'(items.head, right_tail);
end if;
end if;
end partition;

function quicksort(unsorted, tail : list) return list is
left, right : list;
begin
if unsorted = null then
return tail;
else
partition(in unsorted.head, in unsorted.tail, out left, out right);
return quicksort(left, new node'(unsorted.head, quicksort(right, tail)));
end if;
end quicksort;

function sort(unsorted : list) return list is
begin
return quicksort(unsorted, null);
end sort;

2.4.2 Discussion

The quicksort program demonstrates parallel execution derived from both functional composition and parallel composition of statements.

The quicksort function returns a list consisting of the elements of the unsorted list in ascending order, followed by the tail list. The sort function calls the quicksort function with the empty list as the tail list.
If the unsorted list is the empty list, the quicksort function returns the tail list (trivially meeting the specification). Otherwise, the tail of the unsorted list is partitioned into left and right sublists. The left sublist is all elements less than or equal to the head of the unsorted list, and the right sublist is all elements greater than the head of the unsorted list. The quicksort function returns the concatenation of the recursively sorted left sublist, the head of the unsorted list, the recursively sorted right sublist, and the tail list. The tail parameter of the quicksort function provides a method of concatenation.

Since the left and right lists are sublists of the tail of the unsorted list, (if the unsorted list is finite and acyclic) they each have at least one fewer element than the unsorted list. Therefore, the recursion terminates (if the unsorted list is finite and acyclic), because each recursive call sorts a list with fewer elements.

At each level of a sort call, the partition and the two recursive quicksort processes are executed in parallel. In particular, the two recursive quicksort processes are independent of each other and are executed in parallel without interaction. Therefore, the complete executing program consists of a network of parallel quicksort and partition processes.

2.5 Treesort

2.5.1 The Program

```pascal
type node;
type list is access node;
type node is record
  head : integer;
tail : list;
end record;

type tree node;
type tree is access tree node;
type tree node is record
  left : tree;
  value : integer;
  right : tree;
end record;
```
function insert(item : integer; t : tree) return tree is
begin
    if t = null then
        return new tree.node(null, item, null);
    elsif item < t.value then
        return new tree.node(insert(item, t.left), t.value, t.right);
    else
        return new tree.node(t.left, t.value, insert(item, t.right));
    end if;
end insert;

function build(items : list) return tree is
begin
    if items = null then
        return null;
    else
        return insert(items.head, build(items.tail));
    end if;
end build;

function traverse(t : tree; tail : list) return list is
begin
    if t = null then
        return tail;
    else
        return traverse(t.left, new node(t.value, traverse(t.right, tail)));
    end if;
end traverse;

function sort(unsorted : list) return list is
begin
    return traverse(build(unsorted), null);
end sort;

2.5.2 Discussion

The treesort program demonstrates parallel execution derived from functional composition, operating on tree data structures.

Sort: The sort function returns the in-order traversal of an ordered tree built from the unsorted list. The tree is built such that it has the same elements as the unsorted list, and all the elements to the left of each node
are less than the node, and all the elements to the right of each node are
greater than or equal to the node.

The build and traverse processes are executed in parallel. However, the
dependence of the traverse process on the result of the build process means
that their execution can be overlapped only to a small degree.

**Build:** If the items list is the empty list, the build function returns the
empty tree (which is trivially ordered). Otherwise, the build function re-
turns the head of the items list inserted into its correct position in the tree
recursively built from the tail of the items list. The recursion terminates
and builds a finite and acyclic tree (if the items list is finite and acyclic),
because each recursive call builds a tree from a list with one fewer elements.

At each level of a build call, the insert and recursive build processes are
executed in parallel. Therefore, a complete executing build call consists of a
pipeline of parallel build and insert processes. More overlapping of execution
can be obtained in building an ordered tree than in building an ordered list,
because insertions on different branches of the tree are independent of each
other.

**Traverse:** If the tree is the empty tree, the traverse function returns the
tail list. Otherwise, the traverse function returns the concatenation of the
recursively traversed left side of the tree, the root node of the tree, the
recursively traversed right side of the tree, and the tail list. The tail param-
eter of the traverse function (which is the empty list in the outermost call)
provides a method of concatenation. The recursion terminates (if the tree
is finite and acyclic), because each recursive call traverses a tree with fewer
elements.

At each level of a traverse call, the two recursive traverse processes are
independent of each other and are executed in parallel without interaction.
Therefore, a complete executing traverse call consists of a tree of parallel
traverse processes.
2.6 Heapsort

2.6.1 The Program

type node;
  type list is access node;
  type node is record
    head : integer;
    tail : list;
  end record:

type heap node;
  type heap is access heap node;
  type heap node is record
    left : heap;
    left size : integer;
    value : integer;
    right : heap;
    right size : integer;
  end record:

function insert(item : integer; h : heap) return heap is
begin
  if h = null then
    return new heap node(null, 0, item, null, 0);
  elsif item < h.value then
    if h.left size < h.right size then
      return new heap node(insert(h.value, h.left), h.left size + 1,
        item, h.right, h.right size);
    else
      return new heap node(h.left, h.left size, item,
        insert(h.value, h.right), h.right size + 1);
    end if;
  else
    if h.left size < h.right size then
      return new heap node(insert(item, h.left), h.left size + 1,
        h.value, h.right, h.right size);
    else
      return new heap node(h.left, h.left size, h.value,
        insert(item, h.right), h.right size + 1);
    end if;
  end if;
end insert:
function delete_root(h : heap) return heap is  
begin  
    if h.left = null then  
        return h.right;  
    elsif h.right = null then  
        return h.left;  
    elsif h.left.value < h.right.value then  
        return new heap_node(delete_root(h.left), h.left.size - 1,  
            h.left.value, h.right, h.right.size);  
    else  
        return new heap_node(h.left, h.left.size, h.right.value,  
            delete_root(h.right), h.right.size - 1);  
    end if;  
end delete_root;  

function build(items : list) return heap is  
begin  
    if items = null then  
        return null;  
    else  
        return insert(items.head, build(items.tail));  
    end if;  
end build;  

function dismantle(h : heap) return list is  
begin  
    if h = null then  
        return null;  
    else  
        return new node(h.value, dismantle(delete_root(h)));  
    end if;  
end dismantle;  

function sort(unsorted : list) return list is  
begin  
    return dismantle(build(unsorted));  
end sort;  

2.6.2 Discussion

The heapsort program demonstrates parallel execution derived from functional composition, operating on tree data structures.
Sort: The sort function returns a list created by dismantling an ordered and balanced heap built from the unsorted list. The heap is built such that it has the same elements as the unsorted list, all the elements beneath each node are greater than or equal to the node (ordering), and the number of elements to the left and right of each node differ by at most one (balancing).

The dismantle and build processes are executed in parallel. However, the dependence of the dismantle process on the result of the build process means that their execution can be overlapped only to a small degree.

Build: If the items list is the empty list, the build function returns the empty heap (which is trivially ordered and balanced). Otherwise, the build function returns the head of the items list inserted into its correct position in the heap recursively built from the tail of the items list. The recursion terminates and builds a finite and acyclic heap (if the items list is finite and acyclic), because each recursive call builds a heap from a list with one fewer elements.

At each level of a build call, the insert and recursive build processes are executed in parallel. Therefore, a complete executing build call consists of a pipeline of parallel build and insert processes. More overlapping of execution can be obtained in building an ordered heap than in building an ordered list, because insertions on different branches of the heap are independent of each other.

Dismantle: If the heap is the empty heap, the dismantle function returns the empty list. Otherwise, the dismantle function returns a list with as its head the root (minimum) element of the heap, and as its tail the dismantling of the heap formed by deleting the root node (retaining ordering but not necessarily balancing). The recursion terminates (if the heap is finite and acyclic), because each recursive call dismantles a heap with one fewer elements.

At each level of a dismantle call, the delete_root and recursive dismantle processes are executed in parallel. Therefore, a complete executing dismantle call consists of a pipeline of parallel dismantle and delete_root processes. In many cases, processes will operate on different branches of the heap and be independent.
2.7 A Test Program

The following test program provides a general set of tests applicable to any sorting program.

```plaintext
function build_list(size, i1, i2, i3, i4, i5 : integer) return list is
begin
    if size = 0 then
        return null;
    else
        return new node'(i1, build_list(size - 1, i2, i3, i4, i5, 0));
    end if:
end build_list;

procedure put_items(items : in list) is
begin
    sequential
    put(items.head);
    if items.tail /= null then
        sequential
        put(" ");
        put_items(items.tail);
    end if:
end put_items;

procedure put_list(items : in list) is
begin
    if items = null then
        put("empty");
    else
        put_items(items);
    end if:
end put_list;
```
function equal_lists(list_1, list_2 : list) return Boolean is
begin
  if list_1 = null and list_2 = null then
    return true;
  elsif (list_1 = null and list_2 /= null) or
    (list_2 = null and list_1 /= null) then
    return false;
  else
    return list_1.head = list_2.head and then
    equal_lists(list_1.tail, list_2.tail);
  end if;
end equal_lists;

procedure test(message : in string; unsorted, expected : in list) is
  sorted : list;
begin
  sequential
  put(message); new_line;
  put(" Unsorted list: "); put_list(unsorted); new_line;
  sorted := sort(unsorted);
  put(" Sorted list: "); put_list(sorted); new_line;
  if equal_lists(sorted, expected) then
    put("Succeeded");
  else
    put(">>> FAILED <<<");
  end if;
  new_line; new_line;
end test;
procedure main is
  x : integer;
begin
  sequential
  put("Testing Sorting"); new_line;
  put("-------------"); new_line;
  new_line;
  x := 0;
  test("Empty list:",
       build_list(0, x, x, x, x), build_list(0, x, x, x, x));
  test("Single item:",
       build_list(1, 1, x, x, x), build_list(1, 1, x, x, x));
  test("Pair in ascending order:",
       build_list(2, 1, 2, x, x), build_list(2, 1, 2, x, x));
  test("Pair in descending order:",
       build_list(2, 2, 1, x, x), build_list(2, 1, 2, x, x));
  test("Pair with same values:",
       build_list(2, 1, 1, x, x), build_list(2, 1, 1, x, x));
  test("List in ascending order:",
       build_list(5, 1, 2, 2, 3, 4), build_list(5, 1, 2, 2, 3, 4));
  test("List in descending order:",
       build_list(5, 4, 3, 2, 2, 1), build_list(5, 1, 2, 2, 3, 4));
  test("List with same values:",
       build_list(5, 1, 1, 1, 1, 1), build_list(5, 1, 1, 1, 1, 1));
  test("List in random order:",
       build_list(5, 3, 1, 2, 4, 4), build_list(5, 1, 2, 2, 3, 4));
  put("FINISHED."); new_line;
end main;

2.8 Test Output

Testing Sorting
-------------

Empty list:
  Unsorted list: empty
  Sorted list: empty
Succeeded

Single item:
  Unsorted list: 1
  Sorted list: 1
Succeeded
Pair in ascending order:
Unsorted list: 1 2
Sorted list: 1 2
Succeeded

Pair in descending order:
Unsorted list: 2 1
Sorted list: 1 2
Succeeded

Pair with same values:
Unsorted list: 1 1
Sorted list: 1 1
Succeeded

List in ascending order:
Unsorted list: 1 2 2 3 4
Sorted list: 1 2 2 3 4
Succeeded

List in descending order:
Unsorted list: 4 3 2 2 1
Sorted list: 1 2 2 3 4
Succeeded

List with same values:
Unsorted list: 1 1 1 1 1
Sorted list: 1 1 1 1 1
Succeeded

List in random order:
Unsorted list: 3 1 2 4 2
Sorted list: 1 2 2 3 4
Succeeded

FINISHED.

3 Arrays and Loops

Loops provide a convenient method of defining and accessing the components of array data structures. (In fact, because of the single-assignment restriction, the only sensible use of loops is array manipulation.) Parallel
loops can be used to define the components of arrays in parallel. Where data dependencies exist, synchronization occurs automatically through processes suspending when they access undefined components.

In this section we present two programs that use arrays and loops: the all-points shortest-path problem and Dirichlet’s problem (a simple grid problem). As with the sorting programs in the previous section, with appropriate packaging, both programs are correct sequential Ada programs.

For simplicity, the sizes of arrays are defined by constants. In practice, oversized arrays could be declared and only partially used. (A larger subset of Ada could include unconstrained array types and dynamically sized arrays.)

### 3.1 The All-Points Shortest-Path Problem

#### 3.1.1 Problem Specification

A function is required that satisfies the following specification:

```ada
type path_lengths is array(1..N, 1..N) of integer;

function shortest_paths(edges : path_lengths) return path_lengths;
  -- Input Condition:
  --   N ≥ 1, and
  --   for some weighted directed graph G, with vertices V₁, V₂, ..., Vₙ:
  --   ∀ i ∈ 1..N, j ∈ 1..N : edges(i, j) = weight of edge from Vᵢ to Vⱼ, and
  --   G has no cycles of negative length, and
  --   ∀ i ∈ 1..N : edges(i, i) = 0.
  -- Output Condition:
  --   ∀ i ∈ 1..N, j ∈ 1..N :
  --   shortest_paths(edges)(i, j) = length of shortest path from Vᵢ to Vⱼ.
```
3.1.2 The Program

\[ N: \text{constant integer} := 4; \]

\[ \text{type path lengths is array(1..N, 1..N) of integer;} \]
\[ \text{type all path lengths is array(0..N) of path lengths;} \]

\[ \text{function min(x, y: integer) return integer is} \]
\[ \text{begin} \]
\[ \text{if } x < y \text{ then return } x; \text{ else return } y; \text{ end if;} \]
\[ \text{end min;} \]

\[ \text{function shortest paths(edges: path lengths) return path lengths is} \]
\[ \text{paths: all path lengths;} \]
\[ \text{begin} \]
\[ \text{paths(0):= edges;} \]
\[ \text{for } k \text{ in } 1..N \text{ loop} \]
\[ \text{for } i \text{ in } 1..N \text{ loop} \]
\[ \text{for } j \text{ in } 1..N \text{ loop} \]
\[ \text{paths}(k)(i, j):= \text{min}(\text{paths}(k-1)(i, j), \]
\[ \text{paths}(k-1)(i, k) + \text{paths}(k-1)(k, j)); \]
\[ \text{end loop;} \]
\[ \text{end loop;} \]
\[ \text{return paths(N);} \]
\[ \text{end shortest paths;} \]

3.1.3 Discussion

The program to solve the all-points shortest-path problem demonstrates parallel execution derived from parallel loops.

The shortest paths function defines the paths array to be intermediate results satisfying the following specification:

\[ \forall k \in 0..N, i \in 1..N, j \in 1..N : \]
\[ \text{paths}(k)(i, j) = \text{length of shortest path from } V_i \text{ to } V_j \]
\[ \text{with intermediate vertices only in } V_1..V_k. \]

The paths array and the result of the shortest paths function are defined as follows:

- paths(0) = edges.
This is correct because edges(i, j) is the shortest path from \( V_i \) to \( V_j \) with no intermediate vertices.

- \( \forall k \in 1..N, i \in 1..N, j \in 1..N : \)
  
  \[
  \text{paths}(k)(i, j) = \min(\text{paths}(k-1)(i, j), \\
  \text{paths}(k-1)(i, k) + \text{paths}(k-1)(k, j)).
  \]

This is correct by induction on \( k \).

- The shortest paths function returns \( \text{paths}(N) \).

  This is correct because \( \text{paths}(N)(i, j) \) is the shortest path from \( V_i \) to \( V_j \) with no restriction on intermediate vertices.

The \( N \) iterations of each loop statement are executed in parallel. Therefore, the executing program consists of \( N^3 \) parallel assignment statement processes, each defining the value of a different \( \text{paths}(k)(i, j) \). However, the dependence of the value of \( \text{paths}(k)(i, j) \) on the values of \( \text{paths}(k-1)(i, j) \), \( \text{paths}(k-1)(i, k) \), and \( \text{paths}(k-1)(k, j) \) restricts the order and overlapping of the execution of the assignment statements.

### 3.1.4 A Test Program

The following program tests three cases:

1. Four vertices in a line.
2. Four vertices in a ring.
3. Four vertices in a ring with a diagonal.

Testing is limited because of the fixed value of \( N \).
procedure put_paths(paths : in path_lengths) is
begin
  for i in 1 .. N sequential loop
    sequential
    put("");
    for j in 1 .. N sequential loop
      sequential
      put(paths(i, j));
      put(" ");
    end loop:
    new_line;
  end loop:
end put_paths;

function equal(p, q : path_lengths; i, j : integer) return Boolean is
begin
  if i = N and j = N then
    return p(i, j) = q(i, j);
  elsif i < N and j = N then
    return p(i, j) = q(i, j) and equal(p, q, i + 1, 1);
  elsif i <= N and j < N then
    return p(i, j) = q(i, j) and equal(p, q, i, j + 1);
  end if:
end equal;

function equal_paths(p, q : path_lengths) return Boolean is
begin
  return equal(p, q, 1, 1);
end equal_paths;
procedure test(message : in string; edges, expected : in path_lengths) is
  paths : path_lengths;
begin
  sequential
  put(message); new_line;
  put("Edge lengths:"); new_line;
  put_paths(edges);
  put("Shortest paths:"); new_line;
  paths := shortest_paths(edges);
  put_paths(paths);
  if equal_paths(paths, expected) then
    put("Succeeded");
  else
    put(">>>> FAILED <<<<");
  end if;
  new_line; new_line;
end test;

procedure main is
begin
  sequential
  put("Testing All-Points Shortest-Path Problem"); new_line;
  put("-----------------------------------------"); new_line;
  test("Four vertices in a line:",
    path_lengths'((0, 1, 1000, 1000),
    (1, 0, 1, 1000),
    (1000, 1, 0, 1 ),
    (1000, 1000, 1, 0 )),
    path_lengths'((0, 1, 2, 3),
    (1, 0, 1, 2),
    (2, 1, 0, 1),
    (3, 2, 1, 0)));

test("Four vertices in a ring:",
  path_lengths'((0, 1, 8, 1000),
  (1, 0, 1000, 2  ),
  (8, 1000, 0, 4  ),
  (1000, 2, 4, 0  )),
  path_lengths'((0, 1, 7, 3),
  (1, 0, 6, 2),
  (7, 6, 0, 4),
  (3, 2, 4, 0)));

22
3.1.5 Test Output

Testing All-Points Shortest-Path Problem
------------------------------------------

Four vertices in a line:
Edge lengths:
  0 1 1000 1000
  1 0 1 1000
  1000 1 0 1
  1000 1000 1 0
Shortest paths:
  0 1 2 3
  1 0 1 2
  2 1 0 1
  3 2 1 0
Succeeded

Four vertices in a ring:
Edge lengths:
  0 1 8 1000
  1 0 1000 2
  8 1000 0 4
  1000 2 4 0
Shortest paths:
  0 1 7 3
  1 0 6 2
  7 6 0 4
  3 2 4 0
Succeeded
Four vertices in a ring with a diagonal:

Edge lengths:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Shortest paths:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Succeeded

FINISHED.

3.2 Dirichlet’s Problem

3.2.1 Problem Specification

Dirichlet’s problem concerns a two-dimensional grid of cells, each having a numeric value. Cells on the boundary of the grid have constant values over time. Internal cells have values that change over time, beginning with initial values at time $t = 0$. The value of an internal cell at time $t+1$ is a weighted average of the values of the cell and its immediate neighbors at time $t$, given by:

$$cell_{i,j}^{t+1} = (4 \times cell_{i,j}^t + cell_{i-1,j}^t + cell_{i+1,j}^t + cell_{i,j-1}^t + cell_{i,j+1}^t)/8.$$ 

As time progresses, the values of the cells converge to some steady state.

A function is required that satisfies the following specification:
type grid is array(0 .. N + 1, 0 .. N + 1) of float;

function dirichlet(input_grid : grid; iterations : integer) return grid:

— Input Condition:
  — N ≥ 1, and iterations ≥ 0, and
  — input_grid(0, 0) = 0, and input_grid(0, N+1) = 0, and
  — input_grid(N+1, 0) = 0, and input_grid(N+1, N+1) = 0, and
  — input_grid(0, 1..N) = the constant top boundary of a grid, and
  — input_grid(N+1, 1..N) = the constant bottom boundary of a grid, and
  — input_grid(1..N, 0) = the constant left boundary of a grid, and
  — input_grid(1..N, N+1) = the constant right boundary of a grid, and
  — input_grid(1..N, 1..N) = the values of the internal cells of a grid at time t = 0.

— Output Condition:
  — dirichlet(input_grid, iterations)(0, 0..N+1) = input_grid(0, 0..N+1), and
  — dirichlet(input_grid, iterations)(N+1, 0..N+1) = input_grid(N+1, 0..N+1), and
  — dirichlet(input_grid, iterations)(0..N+1, 0) = input_grid(0..N+1, 0), and
  — dirichlet(input_grid, iterations)(0..N+1, N+1) = input_grid(0..N+1, N+1), and
  — dirichlet(input_grid, iterations)(1..N, 1..N) =
    — the values of the internal cells of the grid at time t = iterations.

Figure 1: Grid of cells for Dirichlet’s problem.
3.2.2 The Program

\[ N : \text{constant integer} := 4; \]

\text{type grid is array(0..N+1, 0..N+1) of float;}\

\text{function dirichlet(input grid : grid; iterations : integer) return grid is}\
\text{new grid : grid;}\
\text{begin}\
\text{if iterations = 0 then}\
\text{return input grid;}\
\text{else}\
\text{for i in 0..N+1 loop}\
\text{for j in 0..N+1 loop}\
\text{if i = 0 or i = N + 1 or j = 0 or j = N + 1 then}\
\text{new grid(i, j) := input grid(i, j);}\
\text{else}\
\text{new grid(i, j) := (4.0 * input grid(i, j) + input grid(i - 1, j) + input grid(i + 1, j) + input grid(i, j - 1) + input grid(i, j + 1) ) / 8.0;}\
\text{end if;}\
\text{end loop;}\
\text{end loop;}\
\text{return dirichlet(new grid, iterations - 1);}\
\text{end if;}\
\text{end dirichlet;}\

3.2.3 Discussion

This program to solve Dirichlet’s problem demonstrates parallel execution derived from parallel loops and functional composition. Recursion creates a succession of intermediate grids, without the explicit declaration of an array of grids.

If the number of iterations required is zero, the dirichlet function returns the input grid. Otherwise, the new grid is defined to be the input grid after one iteration, and the dirichlet function returns the result of one fewer iterations recursively applied to the new grid. The recursion terminates (if the number of iterations required is non-negative), because each recursive call requires one fewer iterations.

The N+2 iterations of each loop statement are executed in parallel. Therefore, each level of a dirichlet call consists of \((N+2)^2\) independent paral-
labeled assignment statement processes, each defining the value of a different
`new_grid(i, j)`.

The recursive Dirichlet call is executed in parallel with the calling process.
Therefore, the complete executing program consists of a pipeline of parallel
Dirichlet processes. However, the dependence of the value of `new_grid(i, j)`
on the values of the surrounding components of the input grid restricts the
order and overlapping of the execution of assignment statements between
different levels of recursion.

### 3.2.4 A Test Program

The following program tests three cases:

1. All boundary cells have zero value and all internal cells have the same
   positive initial value.

2. Top boundary cells have the same positive value, bottom boundary
   cells have zero value, side boundary cells have values varying smoothly
   from top to bottom, and all internal cells have zero initial value.

3. Boundary cells have values varying smoothly from a positive value at
   one corner down to zero at the opposite corner, and all internal cells
   have zero initial value.

In each case, 100 iterations are performed. The internal cell values are
expected to reach a steady state consistent with the boundary cell values.
procedure put_grid(g : in grid) is
begin
    for i in 0 .. N + 1 sequential loop
        put(" ");
    end loop;
    for j in 0 .. n + 1 sequential loop
        put(g(i, j));
        put(" ");
    end loop;
    new_line;
end put_grid:

procedure test(initial_grid : in grid; num_iterations : in integer) is
begin
    sequential
    put("Initial grid:"); new_line;
    put_grid(initial_grid);
    put("After "); put(num_iterations); put(" iterations:"); new_line;
    put_grid(dirichlet(initial_grid, num_iterations));
    new_line;
end test:

procedure main is
begin
    sequential
    put("Testing Dirichlet’s Problem"); new_line;
    put("———"); new_line;
    new_line;
    test(grid’((0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
               (0.0, 9.0, 9.0, 9.0, 9.0, 0.0, 0.0),
               (0.0, 9.0, 9.0, 9.0, 9.0, 0.0, 0.0),
               (0.0, 9.0, 9.0, 9.0, 9.0, 9.0, 0.0),
               (0.0, 9.0, 9.0, 9.0, 9.0, 0.0, 0.0),
               (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)),
               100):

test(grid'(0.0, 5.0, 5.0, 5.0, 5.0, 0.0),
   (4.0, 0.0, 0.0, 0.0, 0.0, 4.0),
   (3.0, 0.0, 0.0, 0.0, 0.0, 3.0),
   (2.0, 0.0, 0.0, 0.0, 0.0, 2.0),
   (1.0, 0.0, 0.0, 0.0, 0.0, 1.0),
   (0.0, 0.0, 0.0, 0.0, 0.0, 0.0)),
100);

test(grid'(0.0, 9.0, 8.0, 7.0, 6.0, 0.0),
   (9.0, 0.0, 0.0, 0.0, 0.0, 4.0),
   (8.0, 0.0, 0.0, 0.0, 0.0, 3.0),
   (7.0, 0.0, 0.0, 0.0, 0.0, 2.0),
   (6.0, 0.0, 0.0, 0.0, 0.0, 1.0),
   (0.0, 4.0, 3.0, 2.0, 1.0, 0.0)),
100);

put("FINISHED."); newline;
end main;

3.2.5 Test Output

Testing Dirichlet's Problem
--------------------------

Initial grid:

0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

After 100 iterations:

0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000206 0.000334 0.000334 0.000206 0.000000
0.000000 0.000334 0.000540 0.000540 0.000334 0.000000
0.000000 0.000334 0.000540 0.000540 0.000334 0.000000
0.000000 0.000206 0.000334 0.000334 0.000206 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
4 Streams and Processes

In the previously presented programs, there is no feedback between the production and consumption of data, and statements are ordered so that production precedes consumption. Therefore, with the addition of appropriate packaging, these Declarative Ada programs are also valid sequential Ada programs. In such programs, parallel execution is possible but not necessary.

However, we can write Declarative Ada programs that involve networks of
parallel processes in which there is feedback in the flow of data between producing and consuming processes. In this section we present two such programs: Hamming’s problem and a solution to Dirichlet’s problem that explicitly creates a process for every cell in the grid. Neither of these programs can have their statements reordered to make them correct sequential Ada programs. For both programs, parallel execution is necessary.

Communication within networks of parallel processes is often by means of shared lists. Synchronization occurs automatically through processes suspending when they access undefined elements. Lists used for communication between parallel processes are often called streams.

4.1 Hamming’s Problem

4.1.1 Problem Specification

A function is required that satisfies the following specification:

```ada
type node; type list is access node; type node is record
  head : integer; tail : list; end record;

function hamming(n : integer) return list:
  -- Input Condition:
  -- n >= 0.
  -- Output Condition:
  -- hamming(n) = a list, in ascending order, of the first n integers of the
  -- form 2^i x 3^j x 5^k, for all integers i >= 0, j >= 0, k >= 0.
```

4.1.2 The Program

```ada
type node; type list is access node; type node is record
  head : integer; tail : list; end record;
```
function multiply(n : integer; numbers : list) return list is
begin
  if numbers = null then
    return null;
  else
    return new node'(n * numbers.head, multiply(n, numbers.tail));
  end if;
end multiply;

function merge(input1, input2 : list) return list is
begin
  if input1 = null then
    return input2;
  elsif input2 = null then
    return input1;
  elsif input1.head < input2.head then
    return new node'(input1.head, merge(input1.tail, input2));
  elsif input2.head < input1.head then
    return new node'(input2.head, merge(input1, input2.tail));
  elsif input1.head = input2.head then
    return new node'(input1.head, merge(input1.tail, input2.tail));
  end if;
end merge;

function truncate(n : integer; numbers : list) return list is
begin
  if n = 0 or numbers = null then
    return null;
  else
    return new node'(numbers.head, truncate(n - 1, numbers.tail));
  end if;
end truncate;

function hamming(n : integer) return list is
begin
  times.2 := multiply(2, result);
  times.3 := multiply(3, result);
  times.5 := multiply(5, result);
  merged := merge(times.2, merge(times.3, times.5));
  result := truncate(n, new node'(1, merged));
  return result;
end hamming;
4.1.3 Discussion

The program to solve Hamming’s problem demonstrates a network of processes executing in parallel, with feedback in the communication between processes. Lists are used as communication streams.

The hamming function returns the result of the network of parallel processes in Figure 2. The derivation and proof of this solution are explained in [2, Chapter 5]. We do not attempt to repeat this explanation.

\[ 2 \]
\[ 3 \]
\[ 5 \]
\[ \text{multiply} \]
\[ \text{merge} \]
\[ \text{truncate} \]
\[ \text{result} \]

**Figure 2**: Process network for Hamming’s problem.

In the preceding program, the parallel processes are initiated through parallel composition of statements. This allows the communication streams to be given names. An alternative is the following program, where the parallel processes are initiated through functional composition:

```
function hamming(n : integer) return list is
  result : list;
begin
  result := truncate(n, new node’(1, merge(multiply(2, result),
                                    merge(multiply(3, result), multiply(5, result)))));
  return result;
end hamming;
```

The choice between composition of statements and functional composition is a subjective matter of preferred style.
4.1.4 A Test Program

The following program tests hamming(0), hamming(1), and hamming(20).

function build_list(size : integer;
   i1, i2, i3, i4, i5, i6, i7, i8, i9, i10,
   i11, i12, i13, i14, i15, i16, i17, i18, i19, i20 : integer) return list is
begin
    if size /= 0 then
        return null;
    else
        return new node'(i1, build_list(size - 1,
            i2, i3, i4, i5, i6, i7, i8, i9, i10, i11,
            i12, i13, i14, i15, i16, i17, i18, i19, i20, 0));
    end if;
end build_list;

procedure put_items(items : in list) is
begin
    sequential
    put(items.head);
    if items.tail /= null then
        sequential
        put(" ");
        put_items(items.tail);
    end if;
end put_items;

procedure put_list(items : in list) is
begin
    if items = null then
        put("empty");
    else
        put_items(items);
    end if;
end put_list;
function equal_lists(list_1, list_2 : list) return Boolean is
begin
  if list_1 = null and list_2 = null then
    return true;
  elsif (list_1 = null and list_2 /= null) or
    (list_2 = null and list_1 /= null) then
    return false;
  else
    return list_1.head = list_2.head and then
    equal_lists(list_1.tail, list_2.tail);
  end if;
end equal_lists;

procedure test(n : in integer; expected : in list) is
  numbers : list;
begin
  sequential
  numbers := hamming(n);
  put("hamming("); put(n); put(" = "); put_list(numbers); new_line;
  if equal_lists(numbers, expected) then
    put("Succeeded");
  else
    put(">>> FAILED <<<");
  end if;
  new_line; new_line;
end test;

procedure main is
begin
  sequential
  put("Testing Hamming's Problem"); new_line;
  put("--------------------------------"); new_line;
  new_line;
  test(0, null);
  test(1, new node('1', null));
  test(20, build_list(20, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12,
    15, 16, 18, 20, 24, 25, 27, 30, 32, 36));
  put("FINISHED."); new_line;
end main;
4.1.5 Test Output

Testing Hamming's Problem
-----------------------------

hamming(0) = empty
Succeeded

hamming(1) = 1
Succeeded

hamming(20) = 1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36
Succeeded

FINISHED.

4.2 Dirichlet's Problem Revisited

4.2.1 Problem Specification

In this section we present another program that solves Dirichlet's problem, as specified in Section 3.2.1.

4.2.2 The Program

N : constant integer := 4;

type grid is array(0 .. N + 1, 0 .. N + 1) of float;

type node;

type stream is access node;

type node is record
  value : float;
  more : stream;
end record;

type streams is array(0 .. N + 1, 0 .. N + 1) of stream;
procedure boundary_cell(
    iterations : in integer;
    fixed_value : in float;
    to_internal : out stream
) is
    more_to_internal : stream;
begin
    if iterations = 0 then
        to_internal := null;
    else
        to_internal := new node'(fixed_value, more_to_internal);
        boundary_cell(in iterations - 1, in fixed_value, out more_to_internal);
    end if;
end boundary_cell;

procedure internal_cell(
    iterations : in integer;
    initial_value : in float;
    result_value : out float;
    from_left, from_right, from_above, from_below : in stream;
    to_left, to_right, to_above, to_below : out stream
) is
    new_value : float;
    more_to_left, more_to_right, more_to_above, more_to_below : stream;
begin
    if iterations = 0 then
        result_value := initial_value;
        to_left := null; to_right := null; to_above := null; to_below := null;
    else
        to_left := new node'(initial_value, more_to_left);
        to_right := new node'(initial_value, more_to_right);
        to_above := new node'(initial_value, more_to_above);
        to_below := new node'(initial_value, more_to_below);
        new_value := (4.0 * initial_value +
                      from_left.value + from_right.value +
                      from_above.value + from_below.value) / 8.0;
        internal_cell(in iterations - 1, in new_value, out result_value,
                      in from_left.more, in from_right.more,
                      in from_above.more, in from_below.more,
                      out more_to_left, out more_to_right,
                      out more_to_above, out more_to_below);
    end if;
end internal_cell;
function dirichlet(input_grid : grid; iterations : integer) return grid is
result : grid;
to_left, to_right, to_above, to_below : streams;
begin
for i in 1 .. N loop
  boundary_cell(in iterations, in input_grid(0, i), out to_below(0, i));
  boundary_cell(in iterations, in input_grid(N + 1, i), out to_above(N + 1, i));
  boundary_cell(in iterations, in input_grid(i, 0), out to_right(i, 0));
  boundary_cell(in iterations, in input_grid(i, N + 1), out to_left(i, N + 1));
end loop;
for i in 0 .. N + 1 loop
  for j in 0 .. N + 1 loop
    if i = 0 or i = N + 1 or j = 0 or j = N + 1 then
      result(i, j) := input_grid(i, j);
    else
      internal_cell(in iterations, in input_grid(i, j), out result(i, j),
                    in to_right(i, j - 1), in to_left(i, j + 1),
                    in to_below(i - 1, j), in to_above(i + 1, j),
                    out to_left(i, j), out to_right(i, j),
                    out to_above(i, j), out to_below(i, j));
    end if;
  end loop;
end loop;
return result;
end dirichlet;

4.2.3 Discussion

This program to solve Dirichlet’s problem demonstrates a grid of processes executing in parallel, with two-way communication between neighboring cells. Lists are used as communication streams. Parallel loops and arrays of streams are used to create the grid of processes.

The dirichlet function returns a result grid consisting of the input.grid values for the boundary cells and the results of a grid of internal.cell processes for the internal cells.

There is one internal.cell process for each internal cell in the problem and one boundary.cell process for each boundary cell in the problem. Internal.cell processes communicate values to and from their top, bottom, left, and right neighboring processes. Boundary.cell processes communicate values to their
neighboring internal_cell processes.

The parallel processes are initiated through parallel loops. One loop initiates the boundary_cell processes. Nested loops initiate the internal_cell processes. Two-dimensional arrays of streams provide a means of addressing the shared communication streams during process initiation.

4.2.4 Testing

The program is tested using the same program as in Section 3.2.4. Exactly the same output is produced as in Section 3.2.5.

5 Dining Philosophers

Declarative Ada programs are deterministic: the execution of a program with the same input values will always perform the same computations and assign the same intermediate and output values. If a program executes to
termination, it will always execute to termination. If a program suspends
without termination, it will always suspend without termination. If a pro-
gram terminates with an execution error, it will always terminate with an
execution error. Such repeatability is generally considered to be desirable,
particularly in program development, testing, and debugging.

However, some problems require a nondeterministic program that interacts
with an inherently nondeterministic environment. Since Declarative Ada has
no nondeterministic features we propose the use of a fair merge procedure
from outside of the pure language. The fair merge procedure is described in
[7, Section 5] and [3, Chapter 14].

In this section we present a distributed solution to the dining philosophers
problem. The environment is a collection of independent processes that
each repeatedly request and relinquish access to a shared resource. A con-
trol program grants access to the resource, subject to restrictions on which
processes can have access simultaneously. Nondeterminism is necessary in
the program, to provide fair and efficient access amongst the competing
processes.

The specification of the dining philosophers problem and the method of
solution that we implement are discussed in [2, Chapter 12], using the non-
deterministic UNITY language. However, in our solution, nondeterminism
is isolated in the use of the fair merge procedure. All other components of
our solution are deterministic.

5.1 Problem Specification

5.1.1 General Specification

This version of the dining philosophers problem concerns a static network
of processes, where the processes correspond to the vertices of a finite, undi-
rected, connected graph. The processes are traditionally called philosophers.
Each philosopher is in one of three states: thinking, hungry, or eating. The
only transitions are from thinking to hungry, hungry to eating, and eating
to thinking.

Each philosopher controls its own transitions from thinking to hungry and
from eating to thinking. These transitions must satisfy the following re-
Figure 4: Philosopher state transitions.

A philosopher must not eat forever, i.e., a philosopher in the eating state must eventually choose to undergo the transition from eating to thinking.

An operating system controls the transitions from hungry to eating. These transitions must satisfy the following requirements:

1. A philosopher must not eat forever, i.e., a philosopher in the eating state must eventually choose to undergo the transition from eating to thinking.

2. Philosophers that are neighbors (i.e., directly connected in the underlying graph) cannot be in the eating state simultaneously.

3. A philosopher must not remain hungry forever, i.e., a philosopher in the hungry state must eventually be permitted to undergo the transition from hungry to eating.

For instance, with the graph in Figure 5, philosophers 2 and 3 can be in the eating state simultaneously, but no other philosopher can be in the eating state simultaneously with either of philosophers 1 or 4.

A solution to the dining philosophers problem is an operating system that satisfies requirements 2 and 3 if all philosophers satisfy requirement 1.

5.1.2 Specification of a Distributed System

We require a distributed solution to the dining philosophers problem. A centralized operating system controlling the hungry to eating transitions of
all processes could be a bottleneck in execution, particularly if the network is large and physically distributed, and transitions are frequent. Instead, we split each philosopher into two components: a user process that we call the client, and an operating system process that we call the server.

Client processes control the thinking to hungry and eating to thinking transitions. Server processes control the hungry to eating transitions. Each client process communicates only with its server process. Each server process communicates with its client process and also with neighboring server processes.

The server processes are identical processes that cooperate to provide distributed control of the hungry to eating transitions of all the philosophers. The client processes each represent only one philosopher, and can be different from one another.

As a solution to the dining philosophers problem we require:

- A defined interface between server processes,
- A defined interface between server and client processes, and
- An implementation of server processes,

such that a network of server and client processes can be created that will
satisfy the specification of the problem.

One solution is presented in the following sections. The derivation and proof of this solution are explained in [2, Chapter 12]. We do not attempt to repeat this explanation.

5.2 The Interface Between Server and Client Processes

The server and client processes have interfaces of the following form:
**type** message **is** record
  sender : integer;
  kind : (thinking, hungry, eating, fork, request);
end record;

**type** node;
**type** stream **is** access node;
**type** node **is** record
  item : message;
  more : stream;
end record;

**procedure** server(
  state : in (thinking, hungry, eating);
  N : in integer;
  ids : in array(1 .. N) of integer;
  forks : in array(1 .. N) of (clean, dirty, not_held);
  requests : in array(1 .. N) of Boolean;
  input : in stream;
  output : out array(0 .. N) of stream
);

**procedure** client(
  from_server : in stream;
  to_server : out stream
);

The meaning of the parameters of a server process are as follows:

1. **state** : **in** (thinking, hungry, eating).
   The state of the client.

2. **N** : **in** integer.
   Neighboring servers are numbered #1 to #N.

3. **ids** : **in** array(1 .. N) of integer.
   Neighboring server #i considers this server to be its neighboring server #ids(i).

4. **forks** : **in** array(1 .. N) of (clean, dirty, not_held).
   forks(i) is the state of the fork shared with neighboring server #i.
5. requests : \textbf{in array}(1 .. N) \textbf{of} Boolean.

requests(i) indicates whether or not the request shared with neighboring server #i is held by this server.

6. input: \textbf{in stream}.

The fair merge of streams of messages from the client and neighboring servers. Messages from the client have sender = 0 and kind \in \{thinking, hungry\}, and messages from neighboring server #i have sender = i and kind \in \{fork, request\}.

7. output : \textbf{out array}(0 .. N) \textbf{of} stream.

output(0) is a stream of messages to the client, and output(1 .. N) are streams of messages to neighboring servers #1 to #N. Messages to the client have sender = 0 and kind = eating, and messages to neighboring server #i have sender = ids(i) and kind \in \{fork, request\}.

There is no global numbering scheme amongst the servers. Different servers can have different numbers of neighboring servers, and each server numbers its neighbors locally. The amount of storage required by each server and the efficiency of its operations depend only on the number of neighboring servers, not on the total number of servers in the system.

Messages between a client and server are as follows:

1. A hungry message from the client to the server indicates that the client has made the transition from thinking to hungry.

2. An eating message from the server to the client indicates that the client can make the transition from hungry to eating.

3. A thinking message from the client to the server indicates that the client has made the transition from eating to thinking.

Messages between neighboring servers are either forks or requests (for forks). Exactly one fork and one request are shared between every pair of neighboring servers. Priorities between pairs of neighboring servers are maintained using the forks, which can be either clean or dirty. A server does not send an eating message to its client unless the server holds the forks for all its neighboring servers.
Initially we require:

1. All clients and servers are in the thinking state.

2. The fork shared between each pair of servers is held by one of the servers, and the corresponding request is held by the other one of the servers.

3. All forks are dirty.

4. The priority graph (with each edge directed toward the server that holds the fork for that edge) is acyclic.
5.3 The Server Program

-- Maximum Number Of Neighbors

max_neighbors : constant integer := 3;

-- Message Kinds

thinking : constant integer := 0;
hungry : constant integer := 1;
eating : constant integer := 2;
fork : constant integer := 3;
request : constant integer := 4;

-- Fork States

clean : constant integer := 0;
dirty : constant integer := 1;
not Held : constant integer := 2;

-- Sender Identification Numbers In Messages To Neighbors

type id_array is array(1 .. max_neighbors) of integer;

-- Streams Of Messages

type message is record
sender : integer;
kind : integer;
end record;

type node;
type stream is access node;
type node is record
item : message;
more : stream;
end record;

type streams is array(0 .. max_neighbors) of stream;

-- Fork and Request Status Arrays

type fork_array is array(1 .. max_neighbors) of integer;
type request_array is array(1 .. max_neighbors) of Boolean;
-- Send A Message

procedure send(output : out streams; more_output : in streams; N : in integer; destination : in integer; sender : in integer; kind : in integer ) is
begin
  for i in 0 .. N loop
    if i /= destination then
      output(i) := more_output(i);
    else
      output(i) := new node'(message'(sender, kind), more_output(i));
    end if;
  end loop:
end send;

-- Fork Status Array Containing All Dirty Forks

function dirty_forks(N : integer) return fork_array is
  forks : fork_array;
begin
  for i in 1 .. N loop
    forks(i) := dirty;
  end loop:
  return forks;
end dirty_forks;

-- Are All The Forks Held?

function holds_all(N : integer; forks : fork_array) return Boolean is
begin
  if N = 0 then
    return true;
  elsif forks(N) = not held then
    return false;
  else
    return holds_all(N - 1, forks);
  end if;
end holds_all;
function update_forks(N : integer; 
forks : fork_array; 
id : integer; 
fork_state : integer ) return fork_array is 
updated_forks : fork_array; 
begin 
for i in 1..N loop 
if i = id then 
updated_forks(i) := fork_state; 
else 
updated_forks(i) := forks(i); 
end if; 
end loop; 
return updated_forks; 
end update_forks; 

-- Update Request Status Array 

function update_requests(N : integer; 
requests : request_array; 
id : integer; 
request_state : Boolean ) return request_array is 
updated_requests : request_array; 
begin 
for i in 1..N loop 
if i = id then 
updated_requests(i) := request_state; 
else 
updated_requests(i) := requests(i); 
end if; 
end loop; 
return updated_requests; 
end update_requests;
-- Send Forks To All Neighbors Wanting The Fork

procedure send_forks(output : out streams;
  more_output : in streams;
  N : in integer;
  ids : in id_array;
  forks : in fork_array;
  requests : in request_array;
  updated_forks : out fork_array ) is

begin
  output(0) := more_output(0);
  for i in 1 .. N loop
    if requests(i) then
      output(i) := new node'(message'(ids(i), fork), more_output(i));
      updated_forks(i) := not_held;
    else
      output(i) := more_output(i);
      updated_forks(i) := forks(i);
    end if;
  end loop;
end send_forks;

-- Send Requests To All Neighbors Holding The Fork

procedure send_requests(output : out streams;
  more_output : in streams;
  N : in integer;
  ids : in id_array;
  forks : in fork_array;
  requests : in request_array;
  updated_requests : out request_array ) is

begin
  output(0) := more_output(0);
  for i in 1 .. N loop
    if forks(i) = not_held then
      output(i) := new node'(message'(ids(i), request), more_output(i));
      updated_requests(i) := false;
    else
      output(i) := more_output(i);
      updated_requests(i) := requests(i);
    end if;
  end loop;
end send_requests;
-- The Server

procedure server(
    state : in integer;  -- Thinking, hungry, or eating.
    N    : in integer;  -- Number of neighbors.
    ids  : in id_array; -- Sender id#s in messages to neighbors.
    forks : in fork_array; -- State of forks shared with neighbors.
    requests : in request_array; -- State of requests shared with neighbors.
    input : in stream;  -- Merged messages from client and neighbors.
    output : out streams -- Message streams to client and neighbors.
) is

-- Invariants:
-- 1. state = thinking \Rightarrow
--   \forall i \in 1..N : (forks(i) = dirty and not requests(i)) or
--   (forks(i) = not_held and requests(i)).
-- 2. state = hungry \Rightarrow
--   (\exists i \in 1..N : forks(i) = not_held) and
--   (\forall i \in 1..N : forks(i) = clean or (forks(i) \neq clean and not requests(i))).
-- 3. state = eating \Rightarrow
--   \forall i \in 1..N : forks(i) = dirty.
-- Therefore:
-- 4. (\exists i \in 1..N : forks(i) = clean) \Rightarrow state = hungry.

more_output : streams;
more_output : streams;
updated_forks : fork_array;
updated_requests : request_array;

begin

if state = thinking and input.item.kind = hungry then
    if holds_all(N, forks) then
        send(output, in more_output, in N, in 0, in 0, in eating);
        server(in eating, in N, in ids, in forks, in requests,
               in input.more, out more.output);
    else
        send_requests(output, in more_output,
                      in N, in ids, in forks, in requests, out updated_requests);
        server(in hungry, in N, in ids, in forks, in updated_requests,
               in input.more, out more.output);
    end if;
end if;

51
elsif state = thinking and input.item.kind = request then
  send(out output, in more output,
       in N, in input.item.sender, in ids(input.item.sender), in fork);
  server(in thinking, in N, in ids,
         in update_forks(N, forks, input.item.sender, not held),
         in update_requests(N, requests, input.item.sender, true),
         in input.more, out more output);
elsif state = hungry and input.item.kind = fork then
  updated_forks := update_forks(N, forks, input.item.sender, clean);
  if holds_all(N, updated_forks) then
    send(out output, in more output, in N, in 0, in 0, in eating);
    server(in eating, in N, in ids, in dirty_forks(N), in requests,
           in input.more, out more output);
  else
    server(in hungry, in N, in ids, in updated_forks, in requests,
           in input.more, out output);
  end if;
elsif state = hungry and input.item.kind = request then
  if forks(input.item.sender) = clean then
    server(in hungry, in N, in ids,
           in forks, in update_requests(N, requests, input.item.sender, true),
           in input.more, out output);
  elsif forks(input.item.sender) = dirty then
    send(out output, in more output,
         in N, in input.item.sender, in ids(input.item.sender), in fork);
    send(out more output, in more more output,
         in N, in input.item.sender, in ids(input.item.sender), in request);
    server(in hungry, in N, in ids,
           in update_forks(N, forks, input.item.sender, not held), in requests,
           in input.more, out more more output);
  end if;
elsif state = eating and input.item.kind = thinking then
  send_forks(out output, in more output,
             in N, in ids, in forks, in requests, out updated_forks);
  server(in thinking, in N, in ids, in updated_forks, in requests,
         in input.more, out more output);

\[\text{elsif state} = \text{eating and input.item.kind} = \text{request then} \]
\[\text{server(in eating, in N, in ids,} \]
\[\text{in forks, in update requests(N, requests, input.item.sender, true),} \]
\[\text{in input.more, out output);} \]

end if:

end server;

5.4 Discussion of the Server Program

A server process suspends until an input message is defined. Then, depending on the server state, the kind of the message, and the forks and requests held by the server, one of nine possible cases is deterministically chosen to be executed:

1. State is thinking, a hungry message is received from the client, and all the forks are held.

2. State is thinking, a hungry message is received from the client, and all the forks are not held.

3. State is thinking and a request is received from a neighboring server.

4. State is hungry, a fork is received from a neighboring server, and all the forks are now held.

5. State is hungry, a fork is received from a neighboring server, and all the forks are not yet held.

6. State is hungry, a request is received from a neighboring server, and the fork shared with that neighbor is clean.

7. State is hungry, a request is received from a neighboring server, and the fork shared with that neighbor is dirty.

8. State is eating and a thinking message is received from the client.

9. State is eating and a request is received from a neighboring server.
In each case, appropriate messages are sent in response, and the server
recurses with appropriately modified state, forks, requests, and streams, to
await the next input message. Examination shows that each case maintains
the invariants stated in the header of the server program.

If the language provided a nondeterministic choice construct, we could write
an equivalent program with fewer cases. For example, the four cases in which
a request is received could be combined. However, such a program would
maintain weaker invariants on the server. Therefore, an argument can be
made that the program would be more difficult to reason about.

5.5 A Test Program

The following test program creates the configuration of four philosophers in
Figure 5 and Figure 6. The clients eat and think for an amount of time de-
termined by a busy wait. Each client completes five thinking-hungry-eating
cycles, then terminates in the thinking state. The clients send messages to
a single printer process that outputs a trace of the client state-transitions.
The global numbering of philosophers is only used in producing the output.
The initial distribution of forks and requests gives an acyclic priority graph.

The merger procedure used in the test program cannot be written in De-
clarative Ada. To execute the program, a merger procedure was imported
from PCN [3, 5].

procedure think(time : in integer) is
begin
  if time > 0 then
    think(in time - 1);
  end if;
end think;

procedure eat(time : in integer) is
begin
  if time > 0 then
    eat(in time - 1);
  end if;
end eat;
procedure client(
    id : in integer;
    count : in integer;
    from_server : in stream;
    to_server : out stream;
    to_printer : out stream
) is
    eating_message : message;
begin
    sequential
    to_printer := new node;
    to_printer.item := message'(id, thinking);
    if count > 0 then
        sequential
        think(10);
        to_server := new node;
        to_server.item := message'(0, hungry);
        to_printer.more := new node;
        to_printer.more.item := message'(id, hungry);
        eating_message := from_server.item;
        to_printer.more.more := new node;
        to_printer.more.more.item := message'(id, eating);
        eat(10);
        to_server.more := new node;
        to_server.more.item := message'(0, thinking);
        client(in id, in count - 1, in from_server.more,
            out to_server.more, out to_printer.more.more.more)
    end if;
end client;
procedure printer(messages : in stream) is
begin
  if messages /= null then
    sequential
    put("Philosopher ");
    put(messages.item.sender);
    if messages.item.kind = thinking then
      put(" starts thinking");
    elsif messages.item.kind = hungry then
      put(" becomes hungry");
    elsif messages.item.kind = eating then
      put(" starts eating");
    end if;
    newline:
    printer(in messages.more);
  end if:
end printer;

procedure merger(input_1, ..., input_n : in stream; output : out stream):
  -- Fair-merge procedure imported from outside of the language.

procedure main is
  client1.out, client2.out, client3.out, client4.out : stream;
  client1.print, client2.print, client3.print, client4.print : stream;
  server1.out, server2.out, server3.out, server4.out : streams;
  printer.in : stream;
begin
  client(in 1, in 5, in server1.out(0), out client1.out, out client1.print);
  client(in 2, in 5, in server2.out(0), out client2.out, out client2.print);
  client(in 3, in 5, in server3.out(0), out client3.out, out client3.print);
  client(in 4, in 5, in server4.out(0), out client4.out, out client4.print);

  merger(in client1.print, in client2.print, in client3.print, in client4.print,
        out printer.in);

  printer(in printer.in);

end main;
server(in thinking, in 3, in id_array'(1, 1, 1),
in fork_array'(dirty, dirty, dirty),
in request_array'(false, false, false), in server1_in, out server1_out);
server(in thinking, in 2, in id_array'(1, 2, -1),
in fork_array'(not_held, dirty, not_held),
in request_array'(true, false, false), in server2_in, out server2_out);
server(in thinking, in 2, in id_array'(2, 3, -1),
in fork_array'(not_held, dirty, not_held),
in request_array'(true, false, false), in server3_in, out server3_out);
server(in thinking, in 3, in id_array'(3, 2, 2),
in fork_array'(not_held, not_held, not_held),
in request_array'(true, true, true), in server4_in, out server4_out);

merger(in client1_out, in server2_out(1), in server3_out(1), in server4_out(1),
out server1_in);
merger(in client2_out, in server1_out(1), in server4_out(2),
out server2_in);
merger(in client3_out, in server1_out(2), in server4_out(3),
out server3_in);
merger(in client4_out, in server1_out(3), in server2_out(2), in server3_out(2),
out server4_in);
end main;

5.6 Test Output

Examination of the test output shows that each philosopher completes exactly five thinking-hungry-eating cycles. As we require, no philosopher eats simultaneously with either of philosophers 1 or 4, but as is permitted, philosophers 2 and 3 do eat simultaneously.

Philosopher 1 starts thinking
Philosopher 2 starts thinking
Philosopher 3 starts thinking
Philosopher 4 starts thinking
Philosopher 1 becomes hungry
Philosopher 2 becomes hungry
Philosopher 3 becomes hungry
Philosopher 4 becomes hungry
Philosopher 1 starts eating
Philosopher 1 starts thinking
Philosopher 1 becomes hungry
Philosopher 4 starts eating
Philosopher 4 starts thinking
Philosopher 4 becomes hungry
Philosopher 2 starts eating
Philosopher 3 starts eating
Philosopher 2 starts thinking
Philosopher 3 starts thinking
Philosopher 2 becomes hungry
Philosopher 3 becomes hungry
Philosopher 1 starts eating
Philosopher 1 starts thinking
Philosopher 1 becomes hungry
Philosopher 4 starts eating
Philosopher 4 starts thinking
Philosopher 4 becomes hungry
Philosopher 2 starts eating
Philosopher 3 starts eating
Philosopher 2 starts thinking
Philosopher 3 starts thinking
Philosopher 2 becomes hungry
Philosopher 3 becomes hungry
Philosopher 1 starts eating
Philosopher 1 starts thinking
Philosopher 1 becomes hungry
Philosopher 4 starts eating
Philosopher 4 starts thinking
Philosopher 4 becomes hungry
Philosopher 2 starts eating
Philosopher 3 starts eating
Philosopher 2 starts thinking
Philosopher 3 starts thinking
Philosopher 2 becomes hungry
Philosopher 3 becomes hungry
Philosopher 1 starts eating
Philosopher 1 starts thinking
Philosopher 4 starts eating
Philosopher 4 starts thinking
Philosopher 4 becomes hungry
Philosopher 4 starts thinking
Philosopher 2 starts eating
Philosopher 3 starts eating
Philosopher 2 starts thinking
Philosopher 3 starts thinking

6 Acknowledgment

This research was supported in part by Air Force Office of Scientific Research grant ASFOR-91-0070.

References


