<table>
<thead>
<tr>
<th>Title</th>
<th>On Parallel Tree Traverse in Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Nakao, Zensho; Takeuchi, Kazuo; Yogi, Takashi</td>
</tr>
<tr>
<td>Citation</td>
<td>琉球大学工学部紀要(38): 67-82</td>
</tr>
<tr>
<td>Issue Date</td>
<td>1989-09</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/20.500.12000/1979">http://hdl.handle.net/20.500.12000/1979</a></td>
</tr>
<tr>
<td>Rights</td>
<td></td>
</tr>
</tbody>
</table>
On Parallel Tree Traverse in Ada®

Zensho Nakao*, Kazuo Takeuchi* & Takashi Yogi*

Abstract Two new methods for parallel tree traverse are given, and their implementations are described and tested in Ada which supports parallel programming via Ada tasks. The simulations performed show that (1) when there are no restriction imposed on the number of Ada tasks, the amount of time required for the traverse remains constant (equaling a number related to the height of the tree) for varying number of tree nodes; (2) in case where a pre-assigned number of tasks are used, the traverse time varies inversely as the number of tasks.

O. Preliminary remarks on Ada tasks

Ada tasks are for parallel and real-time programming. An Ada task consists of two parts, i.e., a task specification and a task body, where the former specifies an interface with the rest of the program, and the latter gives an implementation of the objectives of the task. An example of a procedure is given below where two tasks are used; the remarks following "---" symbols should make the list self-explanatory.

```ada
with text_io; use text_io;
procedure TASKTEST is
  task T1; -- task T1 specification
  task T2 is -- task T2 specification
    entry E1; -- entry point of task T2
    entry E2; -- another entry point of task T2
  end task T2;
  task body T1 is -- task T1 body
  begin
    loop
      T2.E1; -- entry call to task T2
      T2. E2; -- another entry call to task T2
    end loop;
```

Received on: May 10,1989.
*Engineering Common Course. #Electrical & Information Engineering
®A preliminary version of this paper was presented at the 40—th Kyushu Regional Joint Meeting of EE—related Societies, Ryukyu Univ., Okinawa, Japan, Oct., 1987.
end T1;
task body T2 is — task T2 body
begin
  loop
    accept E1; — accepts an entry call to E1
    put("Entry E1 is accepted.");
    accept E2 do — accepts an entry call to E2
      put("Entry E2 is accepted.");
    end E2;
  end loop;
end T2;
begin
  null;
end TASKTEST;

1. Traverse methods and evaluations

Given a tree in which no nodes share a common child node, we will consider parallel/concurrent processes which travel all the nodes of the tree. With a sequential method a process can traverse only a subtree at a time, but with parallel methods the processes can traverse the subtrees simultaneously. Here, we will introduce two parallel algorithms.

We will first consider the method in which no restriction on the number of tasks generated is placed. The steps are as follows:

0. To begin with, generate one starting traverse task, (i.e., the task which actually travels along through nodes of a tree) at the root of the tree given. Only traverse tasks should move.

1. Each traverse task reads and display the content of the node where it is at. (It may have to carry out additional work at the node, depending on the nature of the tree to be traced.)

2. Next, each traverse task requests generation of a new child traverse task which behaves exactly the same as the requesting parent task corresponding to each child node except left most child node.

3. Each parent, i.e., old traverse task moves toward the left most node, which is reserved for itself. If no further travel is possible, i.e., the task is at a leaf of the tree, then the task stops its move.

To implement this algorithm into an Ada program, we need one starting traverse task and one generator task which generates traverse tasks at a request of traverse tasks. Upon creation, a new traverse task must be informed on which branching node to go. This information passing can possibly be done via an Ada rendezvous between the old and new tasks. However, a rendezvous between a task and itself is impossible, thus, one generator task is prepared for the purpose, where an old traverse task requests the generator task to create new tasks which continue their traverse further toward the leaves.

In Fig.1 is shown a traverse order of the tree nodes. Note that each task stays at each node for 1.0
second (with a delay statement). Table 1 shows traverse time required. What we find from the results are:

1. The traverse time required is $K \times (\text{tree height}) + 1$ (seconds), where $K (> 1 \text{ sec.})$ denotes the duration of stay at each node (which may be required for processing of information at the node); (2) the number of tasks generated equals the number of leaves; (3) the order of node traverse seems unpredictable.

![Fig. 1 Traverse order of the tree nodes without restriction on the number of tasks](image)

Table 1 Traverse time without restriction on the number of tasks

<table>
<thead>
<tr>
<th>number of branches per node</th>
<th>number of nodes</th>
<th>traverse time (sec)</th>
<th>without a task</th>
<th>with tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>7.30</td>
<td>3.18</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>13.67</td>
<td>3.29</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>21.97</td>
<td>3.40</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>32.40</td>
<td>3.57</td>
<td></td>
</tr>
</tbody>
</table>

(*Each tree has height 2*)
We next consider the second method for tree traverse where a pre-assigned number of Ada tasks are allowed. The steps are as follows:

0. To start with, place into a queue a pointer to the tree root node.
1. Each traverse task picks up a pointer from the queue.
2. Place into the queue all the pointers (to the next nodes) coming out from the node that was indicated by the pointer just picked.
3. Each traverse task first displays the content of the node where it is at (and may do some additional work depending on properties of the tree traversed.)
4. Each traverse task repeats steps 1–3 until the queue becomes empty.

To express this algorithm in Ada, we introduce a fixed number of traverse tasks, a queue task which keeps record of the pointers sent from the traverse tasks, and a semaphore task which keeps the traverse tasks from accessing the queue during steps 1 and 2. In Fig. 2 is a block diagram for the method.

![Fig. 2 Fixed number of traverse tasks in action](image)

We show in Fig. 3 a traverse order in the method. As in the first method, each traverse task stays at each node for 1.0 second. In Table 2 is shown the total traverse time corresponding to varying combinations of the number of tasks and nodes. We find from the table that the the product of the traverse time and the task number equals the node number. Thus, in case the traverse tasks stay at each node for $K(>1)$ seconds, the traverse time is $K \times \text{(node number of the tree)/(task number)}$ seconds. Further, we find that the traverse order of the nodes is fixed; and so the present method is better suited for traversing large trees than the first method, but it is necessary to prepare a large array for the queue task.
Fig. 3 Traverse order of tree nodes with a fixed number of traverse tasks

Table 2. Traverse time with a fixed number of traverse tasks

<table>
<thead>
<tr>
<th>number of branches per node</th>
<th>number of nodes</th>
<th>traverse time (sec)</th>
<th>number of tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7.36</td>
<td>4.22</td>
</tr>
<tr>
<td>3</td>
<td>1 3</td>
<td>13.78</td>
<td>7.52</td>
</tr>
<tr>
<td>4</td>
<td>2 1</td>
<td>22.13</td>
<td>11.69</td>
</tr>
<tr>
<td>5</td>
<td>3 1</td>
<td>32.73</td>
<td>16.91</td>
</tr>
</tbody>
</table>

("Each tree has height 2")
2. Conclusion and remarks

With a usual single processor method (Table 1, without a task or Table 2, with a single task), the traverse time will be at least \( K \times (\text{the number of nodes}) \), where \( K \) is the duration of stay at each node; by the multi-task method (i) without restriction on the number of tasks, the traverse time is given by \( K \times (\text{the tree height} + 1) \), (ii) with a fixed number of tasks, the time required is \( K \times (\text{the number of nodes})/(\text{the number of tasks}) \), reducing the processing time considerably.

The second multi-task method can be used for tree searching problems because the traverse order of the nodes is known (Fig. 3). Further simulation with taller trees should be done to confirm the validity of those formulas obtained.

References


Appendix

— Source programs and packages —

[TT1] · · · · · · · · · · · · without restriction on the number of tasks
TT2] · · · · · · · · · · · · without restriction on the number of tasks (with queue)
[TT3] · · · · · · · · · · · · no task
[TT4] · · · · · · · · · · · · with a fixed number of tasks
[TREE-P] · · · · · · · · · package for constructing trees
[QUEUE-MANAGER] · · · generic package for queues
[TIMER] · · · · · · · · · package for timing
[T-OUT] · · · · · · · · · package for constructing trees (used with TREE-P)
procedure TTI is
package int io is
    type integer is integer;
    procedure TTI is
        type link is access integer;
        root : link := null;
        task vonaker is
            entry create( r : in link; lv : in integer ) do
                sub t := r; level := lv;
            end create;
        end task;
        task type voner is
            type von is access Hori.bodj;
            task bod? von taker is
                von : von;
                sub t : link := null;
                level, id_nuher, r_number : integer := 1;
                begin
                    loop
                        select
                            accept create( r : in link; iv : in integer ) do
                                sub t := r; level := iv;
                            end create;
                            accept search( r : out link; iv, lv : out integer ) do
                                r := sub t; id := id_number; lv := level;
                            end search;
                            accept search( r : out link; id, lv : out integer ) do
                                r := sub t; id := id_number; lv := level;
                            end search;
                            id_number := id_number + 1;
                        or
                            terminate;
                        end select;
                        end loop;
                end loop;
            end task;
            task type wver is
                type wver is access wver_body;
                task wver is
                    entry create( root : in link; iv : in integer ) do
                        sub t := root; id := id_number; lv := level;
                    end create;
                    entry search( r : out link; id, lv : out integer ) do
                        r := sub t; id := id_number; lv := level;
                    end search;
                    entry search( r : out link; id, lv : out integer ) do
                        r := sub t; id := id_number; lv := level;
                    end search;
                    id_number := id_number + 1;
                    or
                    terminate;
                end task;
                end task;
            end task;
        end task;
    end package;
end TTI;
with text io, timer, tree p, queue manager;
use text io, timer, tree p;

procedure TBST is
  package io is new integer io(integer);
  use io;

  type order_queue_element is record
  sub_tree : link;
  level : integer;
  end record;

  limit_queue : constant positive := 512;

  package order_queue is new queue_manager(limit_queue, order_queue_element);
  use order_queue;

  root : link := null;
  dead_number : integer := 1;

  task work maker is
  entry search( r : out link; id, lv : out integer );
  end work maker;

  task type work_body;

  task order is
  entry add_to_queue( r : in link; lv : in integer );
  entry get_from_queue( r : out link; lv : out integer );
  end order;

  type work_type is access work_body;

  ---- task body -------------------------------

  task body work maker is
  entry search( r : out link; id, lv : out integer );
  end work maker;

  task type work_body;

  task order is
  entry add_to_queue( r : in link; lv : in integer );
  entry get_from_queue( r : out link; lv : out integer );
  end order;

  type work_type is access work_body;

  ---- task body -------------------------------

  task body order is
  order_record : order_queue_element;
  begin

  loop
    select
      when queue_full = false =>
        accept add_to_queue( r : in link; lv : in integer ) do
          order_record.sub_tree := r;
          order_record.level := lv;
        end add_to_queue;
        queue(order_record);
    or
      when queue_empty = false =>
        accept get_from_queue( r : out link; lv : out integer ) do
          r := order_record.sub_tree;
          lv := order_record.level;
        end get_from_queue;
    or
      when terminal =>
        terminate;
    end select;
    end loop;
  end order;

  ---- task body -------------------------------

  task body work maker is
  work : work_type;
  sub_tree : link := null;
  level, id_number : integer := 1;
  begin
    loop
      select
        order.get_from_queue( sub_tree, level )
        work := new work_body;
        accept search( r : out link; id, lv : out integer ) do
          r := sub_tree; id := id_number; lv := level;
        end search;
        id_number := id_number + 1;
        else
          error when dead_number /= 1 and id_number = dead_number;
        end select;
      end loop;
    end work maker;
  end work maker;

  ---- task body -------------------------------

  task body work body is
  sub_tree : link := null;
  level, id_number, r_number : integer := 1;
  begin
    order_maker.search( sub_tree, id_number, level )
    put( "MORE F" ); put(id_number, 3); put(" STARTS " ); lap_time;
while sub_tree /= null loop
   put("WARM 1") put(id_number, 1); put(" ");
   for i in sub_tree.range loop
      put( sub_tree.n(i) )
   end loop;
   put(" "); lap_time;
   delay 1.0;
   for i in reverse 1..sub_tree.b'last loop
      if sub_tree.b(i) /= null then
         order.add_to_queue( sub_tree.b(i), level );
         end if;
      end loop;
      level := level + 1;
      sub_tree := sub_tree.b(1);
      end loop;
      dead_warm := dead_warm + 1;
   end loop;
   put("WARM 2") put(id_number, 2); put(" STOPS ");
   end loop;
   end main

begin

  groll( root );

  put_line( "-------------- output of TII -----------------");
  put( "Main routine starts "); timer_start;
  order.add_to_queue( root, 0 );
  put( "Main routine stops "); lap_time;
end TII;
procedure TTJ is
package in_int is new integer_io( integer );
use in_int;
root : link := null;

task body ________________________________
procedure vorm( sub_tree : in link; level : in integer ) is
begin
  put( "CALL TIME = "); lap_time;
  if sub_tree /= null then
    put( "DATA : ");
    for i in 1..sub_tree.a'last loop
      put( sub_tree.a(i) );
    end loop;
    put( " "); lap_time;
    delay 1.0;
    for i in 1..sub_tree.b'last loop
      vorm( sub_tree.b(i), level+1 );
    end loop;
  else
    null;
  end if;
  put( "RETURN TIME = "); lap_time;
  end vorm;

main ________________________________
begin
  vorm( root );
  put( "-------- output of TTJ -------- ");
  put( "Main routine starts : "); timer_start;
  vorm( root, 0 );
  put( "Main routine stops : "); lap_time;

procedure tree is
  package intio is new integer (integer);
  use intio;
  queue_size : constant positive := 5120;
  package qsepio is new queue (size, link);
  use qsepio;
  subtype readers is integer range 1..2;
  root : link;
  task type semaphore is
    entry p;
    entry v;
    end semaphore;
  task type reading_task is
    entry run ( task_id : in readers |)
    end reading_task;
  task link_queue is
    entry push ( sub_tree : in link |)
    entry pop ( sub_tree : out link |
    entry task_stop ( answer : out boolean |)
    end link_queue;
  x : semaphore;
  reader : array ( readers ) of reading_task;
  task body semaphore is
begin
  loop
    select
      accept p;
      or
      accept v;
    end select;
```ada
101:    put( "SHADER" ); putl id_number, 3 ); put( " : " );
102:    for i in sub_tree.a'range loop
103:        put( sub_tree.a(i) );
104:        end loop;
105:        putl( " TIME : " ); log_time;
106:        delay 1.0;
107:        end loop;
108:        s.v;
109:        putl( "SHADER" ); putl id_number, 3 ); putl( " STOPS : " ); log_time;
110:        end reading_task;
111:        begin
112:        grow( root );
113:        put_line( " ---------------- output of TTV ----------------" );
114:        putl( "MAIN ROUTINE STARTS : " ); timer_start;
115:        list_queue.push( root );
116:        for i in readers loop
117:            - reader(i).run(i);
118:            end loop;
119:        putl( "MAIN ROUTINE STOPS : " ); log_time;
120:        end ttv;
```
File name : TREP.P.ADA

---
1: ---------------
2: -- Program : TRE_P
3: --
4: -- ( package )
5: -- Program : T. Takeshii
6: -- Date : July 20, 1987
7: --
8: ---------------
9: package TRE_P is
10: 
11: type NODE( MAX_ELMS : INTEGER := 1;
12: MAX_LINKS : INTEGER := 2 );
13: 
14: type LINK in access NODE;
15: type LINKS in array( INTEGER range 0 ) of LINK;
16: 
17: type RLINKS is array( INTEGER range 0 ) of INTEGER;
18: 
19: type NODE( MAX_ELMS : INTEGER := 1;
20: MAX_LINKS : INTEGER := 2 ) is record
21: N : NODE; X..MAX_ELMS );
22: L : LINKS[ X..MAX_LINKS ] := ( X..MAX_LINKS => NULL );
23: end record;
24: 
25: procedure grow( r : in out link );
26: 
27: end TRE_P;
28: 
29: 
30: 
31: -- Program : TRE_P
32: --
33: -- ( package body )
34: -- Program : T. Takeshii
36: --
37: --
38: 
39: with text_io; use text_io;
40: 
41: package body TRE_P is
42: 
43: procedure grow( r : in out link ) is
44: links_max, e_mail, e_number : integer := 0;
45: begin
46: get( links_max );
47: get( e_mail );
48: get( e_number );
49: if links_max = 0 then
50: r := new node( e_mail, e_number );
51: else
52: r := new node( links_max, e_number );
53: end if;
54: for i in 1..r.a'last loop
55: get( e_number ); r.a[i] := e_number;
56: end loop;
57: if links_max = 0 then
58: r.a[i] := null;
59: else
60: for i in 1.r.a'last loop
61: grow( r.a[i] );
62: end loop;
63: end if;
64: end grow;
65: 
66: end TRE_P;
On Parallel Tree Traverse in Ada: Nakao, Takeuchi & Yosi

File name: QUEUE.ADA
Page: 1

1: ---------------------------------------------------------------
2: PROGRAM QUEUE_MANAGER
3: ---------------------------------------------------------------
4: -- Program designer: M. Nakao
5: -- Language: Ada/Na
6: -- File name: QUEUE.ADA
7: -- Date Ver. 1: June 16th 1987.
8: -- Date Ver. 2: July 30th 1987.
9: ---------------------------------------------------------------
10: generic
11:   length: positive;
12:   type elem is private;
13: package QUEUE_MANAGER is
14:   type message_array is array(1..length) of elem;
15:   procedure queue(text: in elem);  
16:   function enqueue return elem;
17:   function dequeue return boolean;
18:   function queue_empty return boolean;
19:   end QUEUE_MANAGER;
20: ---------------------------------------------------------------
21: PROGRAM QUEUE_MANAGER BODY
22: ---------------------------------------------------------------
23: -- Program designer: M. Nakao
24: -- File name: QUEUE.ADA
25: -- Language: Ada/Na
26: -- Date Ver. 1: June 16th 1987.
27: -- Date Ver. 2: July 30th 1987.
28: ---------------------------------------------------------------
29: package body QUEUE_MANAGER is
30:   type situation is (empty, full, normal);
31:   type := situation := empty;
32:   procedure queue(text: in elem);
33:   function enqueue return elem;
34:   function dequeue return boolean;
35:   function queue_empty return boolean;
36:   procedure queue(text: in elem) in
37:   begin
38:     situation := normal;
39:     message_queue := input point := empty;
40:     input_point := input_point and length + 1;
41:     if input_point := output_point then
42:       situation := full;
43:       end if;
44:       end queue;
45:       end queue;
46:       function enqueue return elem is
47:         put_data := empty;
48:         message_queue := output point := empty;
49:         output_point := output_point and length + 1;
50:         if output_point := input point then
51:           situation := empty;
52:           end if;
53:           return put_data;
54:           end enqueue;
55:           function queue_empty return boolean is
56:             return ( situation := empty );
57:             end queue_empty;
58:             function queue_full return boolean is
59:               return ( situation := full );
60:               end queue_full;
61:               end QUEUE_MANAGER;
62:               #Grad College of Engineering, Osaka Univ.
package TIDES is
    procedure TIIES;
end TIDEs;

package body TIIES is

package body T18BB is
    procedure LI.PJUE ;
end T18BB;

procedure LAP_TIMER is
    ten : constant integer := character'pos'('!');
    s : string[1..11] := ('MiOMW );
    linie : linesize;
    linie, : linesize;
    second : seconds_range;
    hundred : hundreds_range;
begin
    tian.get( tian, minute, second, hundred );
    s[1] := character'val' ( zero + hour / 10 ];
    s[2] := character'val' ( zero + hour % 10 ];
    s[3] := character'val' ( zero + minute / 10 ];
    s[4] := character'val' ( zero + minute % 10 ];
    s[5] := character'val' ( zero + second / 10 ];
    s[6] := character'val' ( zero + second % 10 ];
    s[7] := character'val' ( zero + hundred / 10 ];
    s[8] := character'val' ( zero + hundred % 10 ];
    putl ' 00:00:00:00';
    new_line;
    tian.set( s, 0, 0, 0 );
end TIEIB;
package natural_io is
  type integer_io is access integer;
  task real_io is
  branch : natural := 0;
  height : integer := -1;

procedure out_data( h, b, n : in natural ) is
  procedure put_string( n : in natural ) is
    m : integer := n;
    s : string( 1..6 ) := ( 1..5 => ' ' );
  begin
    if m = 0 then
      s(1) := '0';
    else
      while m /= 0 loop
        l := m * 10;
        m := m / 10;
        s(1) := character'(0..9' => s(1) => character'(0..9' => l mod 10 + character'(0..9' => l div 10); end loop;
    end if;
  end if;
begin
end out_data;
package writer is
  procedure out_data( branch, height, 1 );
end writer;
begin
end writer;