<table>
<thead>
<tr>
<th>項目</th>
<th>内容</th>
</tr>
</thead>
<tbody>
<tr>
<td>タイトル</td>
<td>On Parallel Tree Traverse in Ada</td>
</tr>
<tr>
<td>著者</td>
<td>Nakao, Zensho; Takeuchi, Kazuo; Yogi, Takashi</td>
</tr>
<tr>
<td>引用</td>
<td>琉球大学工学部紀要 (38): 67-82</td>
</tr>
<tr>
<td>発行日</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>
</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;
  end task body T1;
end TASKTEST;
```

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.

![Tree Diagram](image)

**Fig. 1** Traverse order of the tree nodes without restriction on the number of tasks

**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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>without a task</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7.30</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>13.67</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>21.97</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>32.40</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.

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 product of the traverse time and the task number equals the node number. Thus, in case the traverse tasks stay at each node for $K (\geq 1)$ seconds, the traverse time is $K \times \frac{\text{node number of the tree}}{\text{task number}}$ 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>1</td>
<td>2</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 new integer;
  use int.io;
end int.io;

root : link := null;

task von.jaker is
  entry create! r : in link; id : in integer;
  entry search! r : out link; id, lv : out integer;
end von.jaker;

task bodj von taker is
  von : type is access root.type;
  sub.t : link := null;
  level : integer := 0;
  id_number : integer := 1;
begin
  loop
    select
    accept create! r : in link; id : in integer do
      sub.t := r; level := id;
      end create;
    end if;
  end loop;
  level := level + 1;
  for i in reverse sub.t..last loop
    if sub.t.id /= null then
      von.jaker.create! sub.t, id_number, level + 1;
    else
      id_number := id_number + 1;
    end if;
  end loop;
  end if;
end voninator;

task type wora.body; 

end wora.type is access wora.body;

--- task body

--- task body wora.maker is
  sub.t : link := null;
  level : integer := 0;
  id_number : integer := 1;
begin
  loop
    select
    accept create! r : in link; id : in integer | do
      sub.t := r; level := id;
      end create;
    end if;
  end loop;
  super := super.body;
  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 TTI;
with text io, timer, tree p, queue_manager;
use text io, timer, tree p;

procedure TTS is
package io_text is
use io_text;

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_node : integer := 1;

task order is
  entry add_to_queue( r : in link; lv : in integer ) do
    order.record.sub_tree := r;
    order.record.level := lv;
    end add_to_queue;
  entry get_from_queue( r : out link; lv : out integer ) do
    r := order_record.sub_tree;
    lv := order_record.level;
    end get_from_queue;

end order;

task token is
  entry search( r : out link; id, lv : out integer ) do
    r := sub_tree;
    id := id_number;
    lv := level;
    end search;
  entry terminate;

end loop;
end token;

--- task body

begin

loop select
  order.get_from_queue( sub_tree, level );
  token( sub_tree, id_number, level );
  accept search( r : out link; id, lv : out integer ) do
  else
    exit when dead_node /= 1 and id_number = dead_node;
  end loop;
end select;
end loop;
end token;

--- task body

begin

order_record : order_queue_element;
begin

--- task body

begin

put( "TOKEN" ); put( id_number, 3 ); put( "STDOUT" ); lap_time;
while sub_tree /= null loop
  put("WHRN 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_wrn := dead_wrn + 1;
  put("WHRN 2"); put( id_number, 1 ); put(" STOPS :"); lap_time;
  end while_body;

--- main -------------------------------
begin
  sub( root );
  print_line(" -------------- output of T31 -------------");
  put("Main routine starts :"); timer_start;
  order.add_to_queue( root, 0 );
  put("Main routine stops :"); lap_time;
end T32;
with text io, timer, tree p; use text io, timer, tree p;

procedure TT3 is
  package io int is new integer io (integer);
  use io int;

  root : link := null;

  task body -----------------------------
  procedure norm (sub tree : in link; level : in integer) is
    begin
      if sub tree /= null then
        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
          norm( sub tree.b(i), level+1 );
        end loop;
      else
        null;
      end if;
    end norm;

    begin
      norm( root );
      put ( "--------- output of TT3 --------- " );
      put ( "Main routine starts : " ); timer_start;
      norm( root, 0 );
      put ( "Main routine stops : " ); lap_time;
    end TT3;
File name : TTYL.M38  Page : 1

1: 1: ------------------------------
2: 2: - TEST PROGRAM TEST TRACER !
3: 3: - PROGRAM : TTY
4: 4: - ( PROGRAM )
5: 5: - PROGRAMMER : K.Takachi
7: 7: -
8: 8: ------------------------------
9: 9: with text_io,tree,pqueue,manager;
10: use text_io,tree,p;
11: package iotio is new integer(integer);
12: use intio;
13: queue_size: constant positive := 512;
14: package qseio is new queue,iterator(queue.size,link);
15: use qseio;
16: subtype readers is integer range 1..2;
17: root : link;
18: task type semaphore is
19: entry p;
20: entry r;
21: end semaphore;
22: task type reading_task is
23: entry run(task_id : in readers); 24: end reading_task;
25: task link_queue is
26: entry push(sub_tree : in link);
27: entry pop(sub_tree : out link);
28: entry task_stop(answer : out boolean);
29: end link_queue;
30: semaphore;
31: task type semaphore is
32: entry p;
33: entry r;
34: begin
35: loop
36: select
37: accept p;
38: accept r;
39: end select;
40: end loop;
41: 41: or
42: terminate;
43: end select;
44: end loop;
45: end semaphore;
46: task body link_queue is
47: stopped_task : integer := 0;
48: begin
49: loop
50: select
51: when not queue_full ->
52: accept push(sub_tree : in link) do
53: queue(sub_tree);
54: end push;
55: or
56: when not queue_empty ->
57: accept pop(sub_tree : out link) do
58: sub_tree := queue;
59: end pop;
60: or
61: accept task_stop(answer : out boolean) do
62: answer := queue_empty;
63: end task_stop;
64: or
65: terminate;
66: end select;
67: end loop;
68: end link_queue;
69: task body reading_task is
70: sub_tree : link;
71: id_number : readers := 1;
72: stop_ok : boolean := false;
73: begin
74: accept run(task_id : in readers) do
75: id_number := task_id;
76: run;
77: put("READY "); put(id_number, 3); put(" STARTS ": i); loop_time;
78: loop
79: s.p;
80: link_queue.task_stop(stop-ok);
81: exit when stop-ok;
82: link_queue.pop(sub_tree);
83: for i in sub_tree.range loop
84: if sub_tree.B[i] /= null then
85: link_queue.push(sub_tree.B[i]);
86: end if;
87: end loop;
88: s.p;
File name : TTV.ADA

101:    put( "ERROR" ); put( id_number, 3 ); put( " : " );
102:    for i in sub_tree.a.range loop
103:        put( sub_tree.a(i) );
104:        end loop;
105:    put( " TIME : " ); lag_time;
106:    delay 1.0;
107:    end loop;
108:    s.v;
109:    put( "ERROR" ); put( id_number, 3 ); put( " STOPS : " ); lag_time;
110:    end reading_task;
111:
112:    begin
113:    grow( root );
114:    put_line( " " );
115:    put_line( "------------ output of TTV --------------" );
116:    put( "MAIN ROUTINE STARTS : " ); timer_start;
117:    list Queue.push( root );
118:    for i in readers loop
119:        reader( i ).run( i );
120:    end loop;
121:    put( "MAIN ROUTINE STOPS : " ); lag_time;
122:
123:    end tti;
--- Program : TRE_P --
11: --
12: ( package ) --
13: --
15: --
16: --
17: --
18: --
19: package TRE_P is
20:
21: type NODE( MAX_LINKS : INTEGER := 1; MAX_LINKS : INTEGER := 2 );
22:
23: type LINK in access NODE;
24: type LINKS is array( INTEGER range ) of LINK;
25:
26: type LINKS is array( INTEGER range ) of INTEGER;
27:
28: type NODE( MAX_LINKS : INTEGER := 1; MAX_LINKS : INTEGER := 2 ) is record
29: M : LINKS[ 1..MAX_LINKS ];
30: L : LINKS[ 1..MAX_LINKS ] := ( 1..MAX_LINKS ) => NULL ;
31: end record;
32:
33: procedure grow( r : in out link );
34:
35: end TRE_P;
36:
37: ---
38: --- Program : TRE_P --
39: --
40: ( package body ) --
41: --
43: --
44: --
45: with text_io; use text_io;
46: package file is
47: package io_int is new integer_io( integer );
48: use io_int;
49:
50: procedure grow( r : in out link ) is
51:      links_max, clear_max, e_number : integer := 0;
52: begin
53: get( links_max );
54: get( clear_max );
55: if links_max = 0 then
56:      r := new node( clear_max, e_number );
57: end if;
58: for i in 1..r'last loop
59:      get( e_number ); r.r(i) := e_number;
60: end loop;
61:    if links_max = 0 then
62:      r.M(i) := null;
63:    else
64:      for i in 1..r'last loop
65:          grow( r.b(i) );
66:      end loop;
67:      if links_max = 0 then
68:      end if;
69:    end if;
70: end iffl.P;
71:
72: else
80 On Parallel Tree Traverse in Ada: Nakao, Takeuchi & Yoon

File name: QUEUE.ADA

1: -------------------------------
2: -- PROGRAM QUEUE MANAGER --
3: -------------------------------
4: -- Program designer : M. Nakao ----
5: -- Language : Ada ----
6: -- File name : QUEUE.ADA ----
7: -- Date Ver. 1 : June 11th 1987. ----
8: -- Version 2 : July 10th 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( msg : in elem ) ;
16:   function enqueue_return elem ;
17:   function dequeue_empty return boolean ;
18:   function dequeue_full return boolean ;
19: end QUEUE_MANAGER ;
20:   begin
21:     if inputpoint = outputpoint then
22:       put_data : elem ;
23:       return put_data ;
24:     else
25:       put_data := message_queue( input_point ) ;
26:       output_point := output_point and length + 1 ;
27:       if input_point = output_point then
28:         STI := full ;
29:         end if ;
30:       end if ;
31:       return put_data ;
32:       end if ;
33:       put_data := message_queue( output_point ) ;
34:       output_point := input_point and length + 1 ;
35:       if output_point = input_point then
36:         STI := empty ;
37:         end if ;
38:       end if ;
39:       return put_data ;
40:     end if ;
41:     message_queue( input_point ) := msg ;
42:     input_point := positive := 1 ;
43:     end queue_empty ;
44:     function queue_empty return boolean is
45:       return ( STI = empty ) ;
46:     end queue_empty ;
47:     function queue_full return boolean is
48:       return ( STI = full ) ;
49:     end queue_full ;
50:     end body QUEUE_MANAGER ;

Grad College of Engineering, Osaka Univ.
package T18BB is

procedure TIKES

package body T18BB is

package iojot is

s(4) := chracter'val(eero 4 lunite / 10);
55: s(8) := character'val zero 4 second / 10);
procedure t_ool is
  package natural_io is
    use integer_io;
    type futsrl is access integer;
    branch : futsrl := 0;
    height : integer := -1;
  end package;

  procedure out_data( b, k, n : in natural ) is
    procedure put_string( n : in natural ) is
      m : integer := n;
      l : natural := 0;
      c : character := 'z';
      u : string( 1..6 ) := ( 1..6 => ' ' );
    begin
      if m = 0 then
        l := 1;
        u(l) := '0';
      else
        while m /= 0 loop
          l := l + 1;
          u(l) := character mod( m mod 10 + character'pos('0') );
          m := m / 10;
          end loop;
      end if;
      for i in 1..l loop
        c := u(i);
        u(i) := u( i+1 );
        u( i+1 ) := c;
      end loop;
    end put_string;
    begin
      if b = 0 then
        put( "b" );
        put_string(n);
        new_line;
        for i in 1..k loop
          out_data b, k-1, branch;
          end loop;
        end if;
        end out_data;
      else
        put_string(b);
      end if;
  end out_data;
  begin
    begin
      if branch = 0 loop
        get( branch );
        end loop;
      end if;
      while height < 0 loop
        get( height );
        end loop;
      end branch;
      out_data( branch, height, 1 );
    end t_ool;