МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ
КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

ЛАБОРАТОРНА РОБОТА №7

з дисципліни «Паралельні та розподілені обчислення
Паралельне програмування-2»

Тема: «Ada. Рандеву»

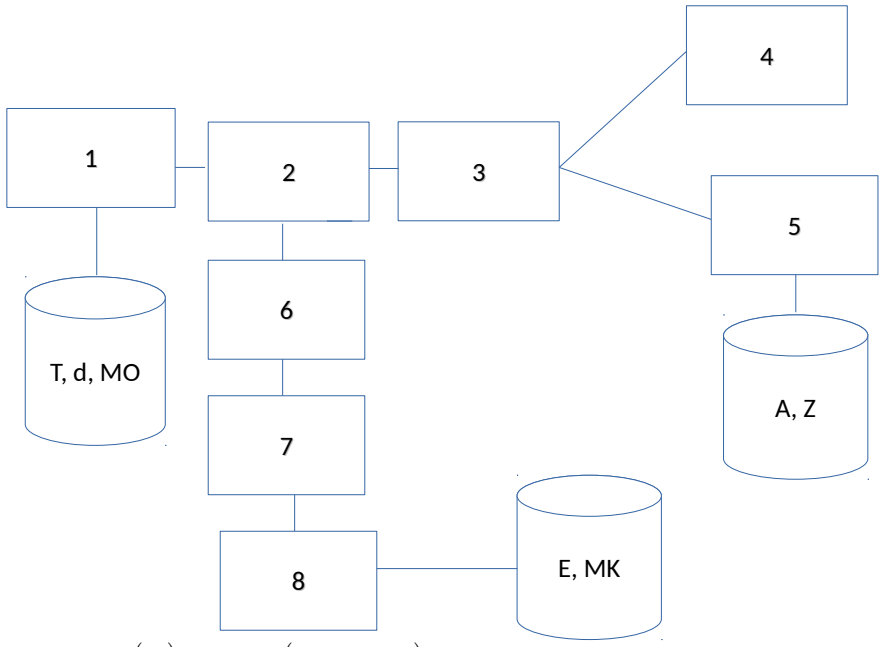
Виконав:
студент 3-го курсу
групи ІП-42
з номер заліковки 4206
Дзюба Влад

Київ – 2017

Завдання

Мета роботи: розробка програми для ПКС з ЛП
Мова програмування: Ada
Засоби організації взаємодії процесів: механізм рандеву.

Варіант



$$MA = max(Z) \cdot E + d \cdot T(MO \cdot MK)$$

Математичний паралельний алгоритм:

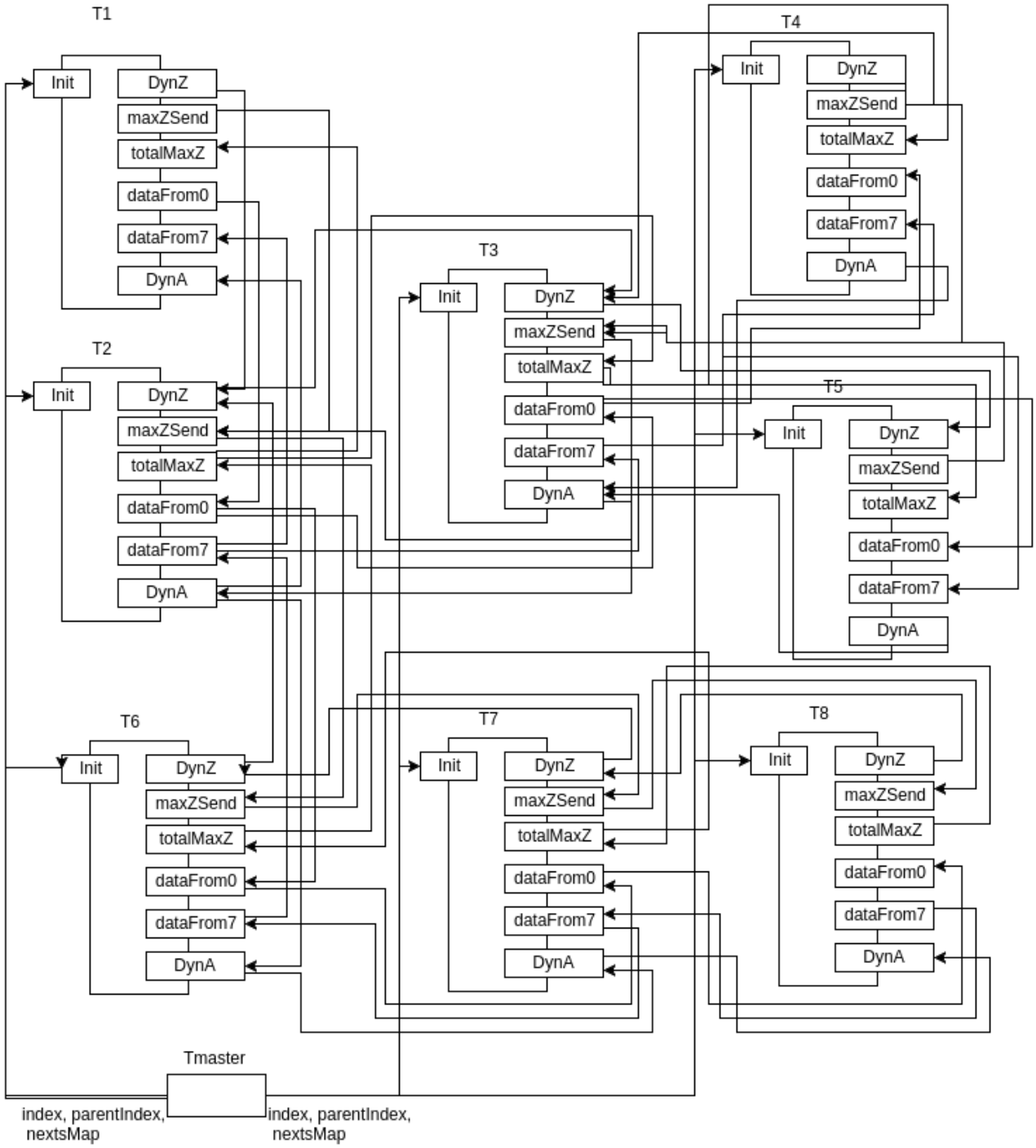
1. $maxZ_i = max(Z_h), i \in [0..P-1]$
2. $$maxZ = max(maxZ_5, max(maxZ_3, maxZ_4, max(maxZ_2, maxZ_1, max(maxZ_6, max(maxZ_7, maxZ_8)))))$$
3. $A_h = maxZ \cdot E_h + d \cdot T(MO \cdot MK_h)$

Алгоритм для кожного процесу:

nextsMap — множина індексів потоків, з яких надходять $maxZ_j$

T_1:		T_5:	
1. Ввести T, d MO.		1. Ввести Z.	
2. Отримати Z_h .	W_1	2. Передати Z_h .	S_1
3. Обчислити $maxZ_1 = max(Z_h)$		3. Обчислити $maxZ_1 = max(Z_h)$	
4. Передати $maxZ_1$	S_1	4. Отримати $maxZ_3$.	W_1
5. Отримати $maxZ$.	S_2	5. Обчислити $maxZ = max(maxZ_4, maxZ_3)$	
6. Передати T, d, MO .	S_3	6. Передати $maxZ$	S_2
7. Отримати E_h, MK_h	W_2	7. Отримати T, d, MO .	W_2
8. Обчислити $A_h = maxZ \cdot E_h + d \cdot T(MO \cdot MK_h)$		8. Отримати E_h, MK_h	W_3
9. Передати A_h	S_3	9. Обчислити $A_h = maxZ \cdot E_h + d \cdot T(MO \cdot MK_h)$	
		10. Отримати A_h	W_4
		11. Вивести A_h	
T_8:		$T_i, i \in \{2,3,4,6,7\}$:	
1. Ввести E, MK.	W_1	1. Отримати Z_h .	W_1
2. Отримати Z_h .		2. Обчислити $maxZ_i = max(Z_h)$	
3. Обчислити $maxZ_8 = max(Z_h)$	W_2	3. Отримати $maxZ_j, j \in nextsMap$.	W_2
4. Передати $maxZ_8$	S_1	4. Обчислити $maxZi = max(maxZ_i, maxZ_j)$ $j \in nextsMap$	
5. Отримати $maxZ$	W_3	5. Передати $maxZ_i$	S_1
6. Отримати T, d, MO .	W_4	6. Отримати $maxZ$	W_3
7. Передати E_h, MK_h	S_2	7. Отримати T, d, MO .	W_4
8. Обчислити $A_h = maxZ \cdot E_h + d \cdot T(MO \cdot MK_h)$		8. Отримати E_h, MK_h	W_5
9. Передати A_h	S_3	9. Обчислити $A_h = maxZ \cdot E_h + d \cdot T(MO \cdot MK_h)$	
		10. Передати A_h	S_2

Структурна схема взаємодії процесів



Лістінг програми:

Main.adb

```
-------------------------------------------------------------------------------
-- main
-- Author:
--   Dzyuba Vlad, IP-42
-- Purpose:
--   Parallel calculating of operations with numbers, vectors and matricies.
-------------------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Hashed_Maps;
with Ada.Containers.Vectors;
use Ada.Containers;

procedure main2 is
  N: constant Integer := 280;
  P: constant Integer := 8;
  H: constant Integer := N / P;
  subtype Index is Integer range 0..N-1;
  subtype PartIndex is Integer range 0..H-1;
  subtype ThreadIndex is Integer range 0..P-1;
  type Vector is array (Index) of Integer;
  type VectorPart is array (PartIndex) of Integer;
  type Matrix is array (Index) of Vector;
  type MatrixPart is array (PartIndex) of Vector;

  function IntHash(id: Integer) return Hash_Type is
  begin
    return Hash_Type(id);
  end;

  package Vectors is new Ada.Containers.Hashed_Maps
    (Key_Type => Integer,
     Element_Type => VectorPart,
     Hash => IntHash,
     Equivalent_Keys => "=");

  package Matrixes is new Ada.Containers.Hashed_Maps
    (Key_Type => Integer,
     Element_Type => MatrixPart,
     Hash => IntHash,
     Equivalent_Keys => "=");

  type DynArray is Array (Integer range <>) of ThreadIndex;
  type PDynArray is access DynArray;

  package Nexts is new Ada.Containers.Hashed_Maps
    (Key_type => ThreadIndex,
     Element_Type => PDynArray,
     Hash => IntHash,
     Equivalent_keys => "=");

  function initVector return Vector is
    res: Vector;
  begin
    for i in Vector'Range loop
      res(i) := 1;
    end loop;
    return res;
  end;

  function initMatrix return Matrix is
    res: Matrix;
  begin
    for i in Matrix'Range loop
      res(i) := initVector;
    end loop;
    return res;
  end;

  task type Calculation is
    entry Init(index, parentIndex: ThreadIndex; nextsMap: Nexts.Map);
    entry DynZ(from: ThreadIndex; vecParts: Vectors.Map);
    entry maxZSend(anotherMaxZ: Integer);
    entry totalMaxZ(totalMaxZ: Integer);
    entry dataFrom0(from: ThreadIndex; T: Vector; d: Integer; MO: Matrix);
    entry dataFrom7(from: ThreadIndex; vecParts: Vectors.Map; matParts: Matrixes.Map);
    entry DynA(from: ThreadIndex; vecParts: Vectors.Map);
  end Calculation;

  calc1, calc2, calc3, calc4, calc5, calc6, calc7, calc8: Calculation;

  function toVectorsMap(vec: Vector) return Vectors.Map is
    res: Vectors.Map;
  begin
    for i in ThreadIndex loop
      declare
        part: VectorPart;
      begin
        for j in PartIndex loop
          part(j) := vec(i*H+j);
        end loop;
        Vectors.Insert(res, i, part);
      end;
    end loop;
    return res;
  end;

  function fromVectorsMap(vecParts: Vectors.Map) return Vector is
    res: Vector;
    procedure addPart(position: Vectors.Cursor) is
      procedure addPart(key: Integer; vecPart: VectorPart) is
      begin
        for i in vecPart'Range loop
          res(key * H + i) := vecPart(i);
```

```
      end loop;
    end;
  begin
    Vectors.Query_Element(position, addPart'Access);
  end;
begin
  Vectors.Iterate(vecParts, addPart'Access);
  return res;
end;

function toMatrixesMap(mat: Matrix) return Matrixes.Map is
  res: Matrixes.Map;
begin
  for i in ThreadIndex loop
    declare
      part: MatrixPart;
    begin
      for j in PartIndex loop
        part(j) := mat(i*H+j);
      end loop;
      Matrixes.Insert(res, i, part);
    end;
  end loop;
  return res;
end;

function remove(vecParts: in out Vectors.Map; index: Integer) return VectorPart is
  res: VectorPart;
begin
  res := Vectors.Element(vecParts, index);
  Vectors.Delete(vecParts, index);
  return res;
end;

function remove(matParts: in out Matrixes.Map; index: Integer) return MatrixPart is
  res: MatrixPart;
begin
  res := Matrixes.Element(matParts, index);
  Matrixes.Delete(matParts, index);
  return res;
end;



task body Calculation is
  index, parentIndex: ThreadIndex;
  nextsMap: Nexts.Map;

  T, Z, E: Vector;
  d, maxZ: Integer;
  MO, MK: Matrix;
  Eh, Zh, Ah: VectorPart;
  MKh: MatrixPart;

begin
  accept Init(index, parentIndex: ThreadIndex; nextsMap: Nexts.Map) do
    Calculation.index := index;
    Calculation.parentIndex := parentIndex;
    Calculation.nextsMap := nextsMap;

    case index is
      when 0 =>
        T := initVector;
        d := 1;
        MO := initMatrix;

      when 4 =>
        Z := initVector;

      when 7 =>
        E := initVector;
        MK := initMatrix;
      when others => null;
    end case;
  end Init;

  if index = 4 then
    declare
      ZParts: Vectors.Map;
    begin
      ZParts := toVectorsMap(Z);
      Zh := remove(ZParts, index);
      calc3.DynZ(index, ZParts);
    end;
  else declare
     fromCopy: Integer;
     vecPartsCopy: Vectors.Map;
     elem: VectorPart;

     procedure sendZhTo(position: Nexts.Cursor) is
       procedure sendZhTo(key: ThreadIndex; neededKeys: PDynArray) is
         cutZh: Vectors.Map;
       begin
         if fromCopy = key then
           return;
         end if;
         for i in neededKeys'Range loop
           elem := Vectors.Element(vecPartsCopy, neededKeys(i));
           Vectors.Insert(cutZh, neededKeys(i), elem);
         end loop;
         case key is
           when 0 => calc1.DynZ(index, cutZh);
           when 1 => calc2.DynZ(index, cutZh);
           when 2 => calc3.DynZ(index, cutZh);
           when 3 => calc4.DynZ(index, cutZh);
           when 4 => calc5.DynZ(index, cutZh);
           when 5 => calc6.DynZ(index, cutZh);
           when 6 => calc7.DynZ(index, cutZh);
```

```
        when 7 => calc8.DynZ(index, cutZh);
       end case;
     end;
   begin
     Nexts.Query_Element(position, sendZhTo'Access);
   end;
 begin
   accept DynZ(from: ThreadIndex; vecParts: Vectors.Map) do
     fromCopy := from;
     vecPartsCopy:= vecParts;
   end;
   Zh := remove(vecPartsCopy, index);
   Nexts.Iterate(nextsMap, sendZhTo'Access);
 end;
end if;
maxZ := Integer'First;
for i in Zh'Range loop
  maxZ := Integer'Max(maxZ, Zh(i));
end loop;

declare
  len: ThreadIndex := Integer(Nexts.Length(nextsMap))-1;
begin
  if parentIndex = index then
    len := len + 1;
  end if;
  for i in 1..len loop
    accept maxZSend(anotherMaxZ: Integer) do
      maxZ := Integer'Max(maxZ, anotherMaxZ);
    end;
  end loop;
end;

if index /= parentIndex then
  case parentIndex is
    when 0 => calc1.maxZSend(maxZ);
    when 1 => calc2.maxZSend(maxZ);
    when 2 => calc3.maxZSend(maxZ);
    when 3 => calc4.maxZSend(maxZ);
    when 4 => calc5.maxZSend(maxZ);
    when 5 => calc6.maxZSend(maxZ);
    when 6 => calc7.maxZSend(maxZ);
    when 7 => calc8.maxZSend(maxZ);
  end case;
  declare
    procedure sendTotalMaxZTo(position: Nexts.Cursor) is
      procedure sendTotalMaxZTo(key: ThreadIndex; neededKeys: PDynArray) is
      begin
        if parentIndex = key then
          return;
        end if;
        case key is
          when 0 => calc1.totalMaxZ(maxZ);
          when 1 => calc2.totalMaxZ(maxZ);
          when 2 => calc3.totalMaxZ(maxZ);
          when 3 => calc4.totalMaxZ(maxZ);
          when 4 => calc5.totalMaxZ(maxZ);
          when 5 => calc6.totalMaxZ(maxZ);
          when 6 => calc7.totalMaxZ(maxZ);
          when 7 => calc8.totalMaxZ(maxZ);
        end case;
      end;
    begin
      Nexts.Query_Element(position, sendTotalMaxZTo'Access);
    end;
  begin
    accept totalMaxZ(totalMaxZ: Integer) do
      maxZ := totalMaxZ;
    end;
    Nexts.Iterate(nextsMap, sendTotalMaxZTo'Access);
  end;
else
  calc3.totalMaxZ(maxZ);
end if;

if index = 0 then
  calc2.dataFrom0(index, T, d, MO);
else
  declare
    fromIndex: ThreadIndex;
    procedure sendDataFrom0To(position: Nexts.Cursor) is
      procedure sendDataFrom0To(key: ThreadIndex; neededKeys: PDynArray) is
      begin
        if fromIndex = key then
          return;
        end if;
        case key is
          when 0 => calc1.dataFrom0(index, T, d, MO);
          when 1 => calc2.dataFrom0(index, T, d, MO);
          when 2 => calc3.dataFrom0(index, T, d, MO);
          when 3 => calc4.dataFrom0(index, T, d, MO);
          when 4 => calc5.dataFrom0(index, T, d, MO);
          when 5 => calc6.dataFrom0(index, T, d, MO);
          when 6 => calc7.dataFrom0(index, T, d, MO);
          when 7 => calc8.dataFrom0(index, T, d, MO);
        end case;
      end;
    begin
      Nexts.Query_Element(position, sendDataFrom0To'Access);
    end;
  begin
    accept dataFrom0(from: ThreadIndex; T: Vector; d: Integer; MO: Matrix) do
      Calculation.T := T;
      Calculation.d := d;
      Calculation.MO := MO;
      fromIndex := from;
    end;
    Nexts.Iterate(nextsMap, sendDataFrom0To'Access);
```

```
  end;
end if;

if index = 7 then
  declare
    EParts: Vectors.Map;
    MKParts: Matrixes.Map;
  begin
    EParts := toVectorsMap(E);
    Eh := remove(EParts, index);
    MKParts := toMatrixesMap(MK);
    MKh := remove(MKParts, index);
    calc7.dataFrom7(index, EParts, MKParts);
  end;
else
  declare
    fromIndex: ThreadIndex;
    vecPartsCopy: Vectors.Map;
    matPartsCopy: Matrixes.Map;
    elem: VectorPart;
    elemMat: MatrixPart;
    procedure sendDataFrom7To(position: Nexts.Cursor) is
      procedure sendDataFrom7To(key: ThreadIndex; neededKeys: PDynArray) is
        cutEh: Vectors.Map;
        cutMKh: Matrixes.Map;
      begin
        if fromIndex = key then
          return;
        end if;
        for i in neededKeys'Range loop
          elem := Vectors.Element(vecPartsCopy, neededKeys(i));
          Vectors.Insert(cutEh, neededKeys(i), elem);
          elemMat := Matrixes.Element(matPartsCopy, neededKeys(i));
          Matrixes.Insert(cutMKh, neededKeys(i), elemMat);
        end loop;
        case key is
          when 0 => calc1.dataFrom7(index, cutEh, cutMKh);
          when 1 => calc2.dataFrom7(index, cutEh, cutMKh);
          when 2 => calc3.dataFrom7(index, cutEh, cutMKh);
          when 3 => calc4.dataFrom7(index, cutEh, cutMKh);
          when 4 => calc5.dataFrom7(index, cutEh, cutMKh);
          when 5 => calc6.dataFrom7(index, cutEh, cutMKh);
          when 6 => calc7.dataFrom7(index, cutEh, cutMKh);
          when 7 => calc8.dataFrom7(index, cutEh, cutMKh);
        end case;
      end;
    begin
      Nexts.Query_Element(position, sendDataFrom7To'Access);
    end;
  begin
    accept dataFrom7(from: ThreadIndex; vecParts: Vectors.Map; matParts: Matrixes.Map) do
      fromIndex := from;
      vecPartsCopy := vecParts;
      matPartsCopy := matParts;
    end;
    Eh := remove(vecPartsCopy, index);
    MKh := remove(matPartsCopy, index);
    Nexts.Iterate(nextsMap, sendDataFrom7To'Access);
  end;
end if;


for i in Ah'Range loop
  declare
    mok: Integer;
  begin
    Ah(i) := maxZ * Eh(i);
    for j in T'Range loop
      mok := 0;
      for k in MO'Range loop
        mok := mok + MO(k)(j) * MKh(i)(k);
      end loop;
      Ah(i) := Ah(i) + d * T(j) * mok;
    end loop;
  end;
end loop;


declare
  gVecParts: Vectors.Map;
  procedure mixin(position: Vectors.Cursor) is
    procedure mixin(key: Integer; vecPart: VectorPart) is
    begin
      Vectors.Insert(gVecParts, key, vecPart);
    end;
  begin
    Vectors.Query_Element(position, mixin'Access);
  end;
  len: Integer := Integer(Nexts.Length(nextsMap))-1;
begin
  Vectors.Insert(gVecParts, index, Ah);
  if index = parentIndex then
    len := len + 1;
  end if;
  for i in 1..len loop
    accept dynA(from: ThreadIndex; vecParts: Vectors.Map) do
      Vectors.Iterate(vecParts, mixin'Access);
    end;
  end loop;
  if index /= parentIndex then
    case parentIndex is
      when 0 => calc1.dynA(index, gVecParts);
      when 1 => calc2.dynA(index, gVecParts);
      when 2 => calc3.dynA(index, gVecParts);
      when 3 => calc4.dynA(index, gVecParts);
      when 4 => calc5.dynA(index, gVecParts);
      when 5 => calc6.dynA(index, gVecParts);
      when 6 => calc7.dynA(index, gVecParts);
```

```
          when 7 => calc8.dynA(index, gVecParts);
        end case;
      else
        declare
          A: Vector := fromVectorsMap(gVecParts);
        begin
          if N < 20 then
            for i in A'Range loop
              put(Integer'Image(A(i)));
            end loop;
            put_line("");
          end if;
        end;
      end if;
    end;

  end Calculation;

begin
  declare
    nextsMap: Nexts.Map;
    keys1: PDynArray;
  begin
    keys1 := new DynArray(0..6);
    for i in keys1'Range loop
      keys1(i) := i + 1;
    end loop;
    Nexts.Insert(nextsMap, 1, keys1);
    calc1.Init(0, 1, nextsMap);
  end;
  declare
    nextsMap: Nexts.Map;
    keys0: PDynArray;
    keys2: PDynArray;
    keys5: PDynArray;
  begin
    keys0 := new DynArray(0..0);
    keys0(0) := 0;
    keys2 := new DynArray(0..2);
    keys2(0) := 2; keys2(1) := 3; keys2(2) := 4;
    keys5 := new DynArray(0..2);
    keys5(0) := 5; keys5(1) := 6; keys5(2) := 7;

    Nexts.Insert(nextsMap, 0, keys0);
    Nexts.Insert(nextsMap, 2, keys2);
    Nexts.Insert(nextsMap, 5, keys5);

    calc2.Init(1, 2, nextsMap);
  end;
  declare
    nextsMap: Nexts.Map;
    keys1: PDynArray;
    keys3: PDynArray;
    keys4: PDynArray;
  begin
    keys1 := new DynArray(0..4);
    keys1(0) := 0; keys1(1) := 1; keys1(2) := 5; keys1(3) := 6; keys1(4) := 7;
    keys3 := new DynArray(0..0);
    keys3(0) := 3;
    keys4 := new DynArray(0..0);
    keys4(0) := 4;

    Nexts.Insert(nextsMap, 1, keys1);
    Nexts.Insert(nextsMap, 3, keys3);
    Nexts.Insert(nextsMap, 4, keys4);

    calc3.Init(2, 4, nextsMap);
  end;
  declare
    nextsMap: Nexts.Map;
    keys2: PDynArray;
  begin
    keys2 := new DynArray(0..6);
    keys2(0) := 0; keys2(1) := 1; keys2(2) := 2;
    keys2(3) := 4; keys2(4) := 5; keys2(5) := 6; keys2(6) := 7;

    Nexts.Insert(nextsMap, 2, keys2);

    calc4.Init(3, 2, nextsMap);
  end;
  declare
    nextsMap: Nexts.Map;
    keys2: PDynArray;
  begin
    keys2 := new DynArray(0..6);
    keys2(0) := 0; keys2(1) := 1; keys2(2) := 2; keys2(3) := 3;
    keys2(4) := 5; keys2(5) := 6; keys2(6) := 7;

    Nexts.Insert(nextsMap, 2, keys2);

    calc5.Init(4, 4, nextsMap);
  end;
  declare
    nextsMap: Nexts.Map;
    keys1: PDynArray;
    keys6: PDynArray;
  begin
    keys1 := new DynArray(0..4);
    keys1(0) := 0; keys1(1) := 1; keys1(2) := 2; keys1(3) := 3; keys1(4) := 4;
    keys6 := new DynArray(0..1);
    keys6(0) := 6; keys6(1) := 7;

    Nexts.Insert(nextsMap, 1, keys1);
    Nexts.Insert(nextsMap, 6, keys6);

    calc6.Init(5, 1, nextsMap);
  end;
  declare
```

```
    nextsMap: Nexts.Map;
    keys5: PDynArray;
    keys7: PDynArray;
  begin
    keys5 := new DynArray(0..5);
    keys5(0) := 0; keys5(1) := 1; keys5(2) := 2; keys5(3) := 3; keys5(4) := 4; keys5(5) := 5;
    keys7 := new DynArray(0..0);
    keys7(0) := 7;

    Nexts.Insert(nextsMap, 5, keys5);
    Nexts.Insert(nextsMap, 7, keys7);

    calc7.Init(6, 5, nextsMap);
  end;
  declare
    nextsMap: Nexts.Map;
    keys6: PDynArray;
  begin
    keys6 := new DynArray(0..6);
    for i in keys6'Range loop
      keys6(i) := i;
    end loop;

    Nexts.Insert(nextsMap, 6, keys6);

    calc8.Init(7, 6, nextsMap);
  end;
end;
```