4. Коноплянко В.И. Организация и безопасность дорожного движения [Текст]: Учебник для вузов / В.И. Коноплянко. – М.: Транспорт, 1991. – 183 с.
5. Єрмак О.М., Сумець А.І. Дослідження факторів, що впливають на небезпеку руху на перехрестях [Текст] / О.М. Єрмак, А.І. Сумець // Комунальне господарство міст. – 2011. – Вип. 101. – С. 284-292.

Abstract

This article concerns the existance of possible processes taking place during the road traffic. The main objective of the research was to determine the possibility of an accident when drivers violate the road regulations. Using the method of field observation, we had obtained division of the number of violations depending on the volume of traffic and hours. The usage of distribution laws of random quantity helped to obtain the regularities adequate to the factual knowledge. The results of the research may be used while estimating the accident rate at crossroads. This permits to forecast the accident rate more precisely on the existing as well as on the designed roads.

Keywords: accident, volume of traffic, possibility

Стаття містить опис можливостей реалізації деяких директив технології OpenMP засобами мови програмування Ада. У роботі наведені різні способи реалізації одного й того ж варіанту директиви та проведено аналіз їхньої ефективності й доцільності. Стаття містить приклади програмного коду

Ключові слова: технологія OpenMP, перетворення програмного коду

Статья содержит описание возможностей реализации некоторых директив технологии OpenMP средствами языка программирования Ада. В работе представлены разные способы реализации одного и того же варианта директивы и проведен анализ их эффективности и целесообразности. Статья содержит примеры программного кода

Ключевые слова: технология OpenMP, преобразования программного кода

УДК 004.451.45, 004.451.24, 519.687.1

ПРО МОЖЛИВІСТЬ РЕАЛІЗАЦІЇ ДИРЕКТИВ ТЕХНОЛОГІЇ OPENMP ДЛЯ МОВИ ПРОГРАМУВАННЯ АДА

О.Г. Лозова*
Контактний тел.: 066-187-99-50
Б.О. Онищенко
Кандидат фізико-математичних, доцент*
Контактний тел.: 066-767-46-49
E-mail: boris@cdu.edu.ua
І.М. Сиволовський
Старший викладач
Кафедра інформаційних систем та медичних технологій**
Контактний тел.: 093-575-40-32
E-mail: grom193@gmail.com
О.О. Супруненко
Кандидат технічних наук, доцент*
Контактний тел.: 066-187-99-50
E-mail: ra-oks@mail.ru
*Кафедра програмного забезпечення автоматизованих систем**
**Черкаський національний університет
ім. Богдана Хмельницького
бул. Шевченка, 81, м. Черкаси, Україна, 18031

1. Вступ

З огляду на те, що останні тенденції в нарощуванні потужностей процесорів персональних комп'ютерів йдуть шляхом збільшення кількості їх ядер, тобто фактично кожен ПК зараз являє собою багатопроцесорну систему, необхідно також використовувати й інші підходи у процесі розроблення ефективно працюючого на таких комп'ютерах програмного забезпечення. Сучасне програмне забезпечення повинно мати можливість використання одночасно кількох ядер (процесорних елементів) комп'ютера, а отже, воно повинно розроблятися з використанням технологій паралельного програмування. Зараз існує

багато технологій розробки паралельних програм. На особливу увагу заслуговують засоби, що роблять можливим швидке «розпаралелення» вже розробленої послідовної програми шляхом додавання до її вихідного коду спеціальних інструкцій, за допомогою яких середовище компіляції сформує паралельний варіант коду та відповідний, паралельно працюючий виконуваний файл. До засобів такого типу належить технологія OpenMP [1, 2].

Технологія OpenMP, яка була розроблена для мов програмування C, C++ та Fortran, виявилась настільки зручною, що було зроблено декілька спроб адаптувати її для інших мов програмування. Зокрема в межах проекту PascalABC.NET реалізовано підтримку найбільш уживаних директив OpenMP [3] в середовище програмування PascalABC.NET. У роботі [4] показано можливість адаптації технології OpenMP для мови програмування Ада.

2. Виділення проблеми, постановка задачі та формування цілей статті

Мова програмування Ада має власні засоби для розроблення паралельного програмного забезпечення. До таких засобів належать задачі (Tasks) та захищені модулі (Protected Units) [5, 6]. За допомогою зазначених мовних конструкцій можна розробити паралельно працюючу програму будь-якої складності. Однак, з іншого боку, використання вбудованих засобів розпаралелювання мови Ада може в окремих випадках надмірно ускладнити процес розробки програмного забезпечення та зробити текст програми недоступним для розуміння та аналізу. До таких випадків можна віднести програмування чисельних алгоритмів, алгоритмів, у яких проводиться обробка великих масивів даних тощо. Більшої зручності при розпаралелюванні таких алгоритмів надають засоби, подібні до технології OpenMP.

У науковій літературі [4] показано, що кожній директиві OpenMP можна поставити у відповідність певну комбінацію мовних конструкцій мови Ада так, що отримана після перетворення програма матиме паралельну структуру, і, відповідно, буде більш ефективно працювати на багатопроцесорних комп'ютерах. Зауважимо, що в зазначеній статті автор використав не всі можливості мови в частині розробки паралельних програм. Приклади, які наводить автор публікації [4], швидше за все відображають можливості попереднього стандарту мови Ада, хоча стаття написана в період активного впровадження нової, більш розвиненої редакції мови. З урахуванням нових можливостей мови Ада, перетворення від послідовної до паралельної програми можна здійснити різними способами, які будуть відрізнятись один від одного як за ефективністю, так і за використанням системних ресурсів.

У цій роботі буде показана можливість введення директив, аналогічних до директив OpenMP, у мову програмування Ада, а також проаналізовано можливі способи заміни директив паралельними мовними конструкціями та представлена загальна послідовність дій під час переходу від послідовної Ада-програми з директивами OpenMP до паралельної Ада-програми.

3. Введення директив OpenMP у мову програмування Ада

Директиви OpenMP – це спеціальним чином оформлені вказівки для компілятора чи іншого модуля, що буде їх обробляти. Вони організовані таким чином, що не впливають на роботу послідовної програми. Зокрема, у мовах C/C++ директиви OpenMP позначено у вигляді вказівок препроцесору:

#pragma omp <директива> [<опція> [[,] <опція>]...]
 <послідовність операторів>;

а в мові Fortran – це спеціального виду коментарі:

!$omp <директива> [<опція> [[,] <опція>]...]
 <послідовність операторів>
!$omp end <директива> [<опція> [[,] <опція>]...].

Усі складні інструкції мови Ада мають початок та обов'язково повинні мати кінець, наприклад, **if** <логічний вираз> **then** <послідовність операторів> **end if**; [5, 6]. Тому більш доцільно було б організувати директиви OpenMP у мові програмування Ада подібно до того, як це зроблено в мові Fortran, тобто за допомогою коментарів спеціального виду:

--omp <директива> [<опція> [[,] <опція>]...]
 <послідовність операторів>
--omp end <директива> [<опція> [[,] <опція>]...].

4. Перехід від директив OpenMP до паралельних конструкцій мови Ада

Розглянемо тепер можливі способи заміни конкретних директив OpenMP на паралельні конструкції мови Ада. Почнемо розгляд з директиви parallel. Наприклад, є програма на мові Ада, що містить директиву parallel:

```
with ada.text_io;
use  ada.text_io;
procedure omp_parallel_demo_1 is
begin
  put_line("Послідовна область 1");
--omp parallel
  put_line("Паралельна область");
--omp end parallel
  put_line("Послідовна область 2");
end omp_parallel_demo_1;
```

Наведена програма в послідовному варіанті просто виведе одне за одним три повідомлення. Якщо ж урахувати директиву parallel, то програма повинна вивести повідомлення *"Послідовна область 1"*, потім декілька разів, залежно від кількості процесорних елементів, вивести повідомлення *"Паралельна область"* і, нарешті, вивести повідомлення *"Послідовна область 2"*.

Перетворимо програму, використовуючи паралельні конструкції мови програмування, для отримання результату з урахуванням директиви parallel. При цьому слід також враховувати те, що кожна паралельна ділянка повинна отримувати свій ідентифікаційний номер, починаючи з номера 0. Один із можливих варіантів такого перетворення наведено нижче:

```
with ada.text_io;
use  ada.text_io;
with system.multiprocessors;
use  system.multiprocessors;
procedure omp_parallel_demo_1 is
begin
```

```
  put_line(“Послідовна область 1”);
--omp parallel
  omp_parallel_1: declare
    N_Threads : CPU := Number_Of_CPUs;
    task type t_par1 is
      entry start(loc_id : in CPU_Range);
    end t_par1;
    task body t_par1 is
      id : integer;
    begin
      accept start(loc_id : in CPU_Range) do
        id := integer(loc_id);
      end start;
      put(“id =” & id’Img);
      put_line(“Паралельна область”);
    end t_par1;
    a_t_par1 : array (0..N_Threads-1) of t_par1;
  begin
    for i in 0..N_Threads-1 loop
      a_t_par1(i).start(i);
    end loop;
  end omp_parallel_1;
--omp end parallel
  put_line(“Послідовна область 2”);
end omp_parallel_demo_1;
```

Щоб визначити, на яку кількість паралельних фрагментів потрібно розбивати програму, необхідно встановити кількість доступних процесорних елементів. Для цього використано функцію Number_Of_CPUs пакету system.multiprocessors, якою ініційовано змінну

N_Threads : CPU := Number_Of_CPUs.

Визначити кількість процесорних елементів можна й іншим способом, а саме за допомогою функції Value(), пакету для роботи зі змінними оточення операційної системи Ada.Environment_Variables, зчитати та перетворити до цілого типу значення змінної оточення number_of_processors:

N_Threads : integer :=
= integer’Value(Value(“number_of_processors”)).

Паралельна робота виділеного директивою parallel фрагменту програми забезпечується за допомогою задач. У блоці **(declare)** omp_parallel_1 оголошено задачний тип t_par1 із входом **(entry)** start. Тіло задачного типу містить паралельний фрагмент програми. Далі в тексті програми оголошено масив a_t_par1 з N_Threads задач типу t_par1, які будуть виконуватись паралельно. За допомогою входу й механізму рандеву кожній задачі передається її ідентифікаційний номер id. До паралельного фрагменту програми додано функцію, що виводить на екран номер задачі:

put(“id =” & id’Img).

Розглянемо й інші способи передавання ідентифікаційного номера в задачу. Один із них – використання дискримінанту в оголошенні задачного типу: **task type** t_par1 (id : CPU_Range). У такому випадку вхід для передавання номера не потрібний, але механізм створення та активації задачі дещо зміниться:

```
...
--omp parallel
  omp_parallel_1: declare
    N_Threads : CPU := Number_Of_CPUs;
    task type t_par1 (id : CPU_Range);
    task body t_par1 is
    begin
      put(“id =” & id’Img);
      put_line(“Паралельна область”);
    end t_par1;
    type p_t_par1 is access t_par1;
    a_p_t_par1 : array (0..N_Threads-1) of p_t_par1;
  begin
    for i in 0..N_Threads-1 loop
      a_p_t_par1(i) := new  t_par1(i);
    end loop;
  end omp_parallel_1;
--omp end parallel
...
```

У цьому випадку оголошено масив вказівників на задачний тип, а самі задачі створюються динамічно, за допомогою оператора **new**. Недоліком даного способу є необхідність звільнення виділених ресурсів пам’яті під час динамічного створення задачі в «ручному» режимі.

Інший спосіб полягає у використанні модуля захищеного типу, який буде містити поточне значення ідентифікаційного номера, передавати його задачі за її запитом та нарощувати. Причому цей модуль можна помістити в окремий пакет, що дозволить покращити зрозумілість основного тексту програми та дасть можливість використовувати один і той же опис для різних паралельних фрагментів, у міру виникнення потреби їхнього використання в програмі:

```
package manager_id is
  protected type id_man_t is
    procedure get_id (loc_id : out integer);
  private
    id : integer := 0;
  end id_man_t;
end manager_id;
package body manager_id is
  protected body id_man_t is
    procedure get_id (loc_id : out integer) is
    begin
      loc_id := id;
      id := id + 1;
    end get_id;
  end id_man_t;
end manager_id;
```

Як видно з фрагменту, у захищеному модулі оголошена приватна змінна id, значення якої може змінюватись лише процедурами самого модуля. Тепер пакет manager_id потрібно підключити до програми, у блоці описати об’єкт типу id_man_t. Новостворена задача може викликати процедуру get_id() захищеного модуля, через яку отримує значення власного ідентифікаційного номера:

```
...
with manager_id;
use  manager_id;
...
--omp parallel
  omp_parallel_1: declare
    N_Threads : CPU := Number_Of_CPUs;
    id_man    : id_man_t;
    task type t_par1;
    task body t_par1 is
    begin
      id_man.get_id(id);
```

```
    put("id =" & id'Img);
    put_line("Паралельна область");
  end t_par1;
  a_t_par1 : array (0..N_Threads-1) of t_par1;
 begin
  null;
 end omp_parallel_1;
--omp end parallel
...
```

Цей спосіб, на нашу думку, є найбільш вдалим, тому що він не потребує організації додаткових циклічних структур. Захищений же модуль займає тільки ресурси в пам'яті комп'ютера і не використовує процесорних ресурсів. До того ж його можна розширити іншими засобами, наприклад, процедурою примусового встановлення нового значення ідентифікаційного номера.

Згідно зі стандартом OpenMP, директива parallel містить ряд додаткових опцій [1]: private(<список>) – визначає список змінних, для яких створюється локальна копія в кожній паралельній ділянці, причому початкове значення цих копій вважається невизначеним; firstprivate(<список>) – визначає список змінних, для яких створюється локальна копія в кожній паралельній ділянці, кожній такій змінній присвоюється значення, яке мала відповідна змінна до директиви parallel; shared(<список>) – визначає список змінних, спільних для всіх паралельних ділянок; reduction(<оператор>:<список>) – визначає список змінних, для яких створюється локальна копія в кожній паралельній ділянці, а після закінчення виконанні паралельного фрагменту до всіх локальних копій змінних застосовується вказаний <оператор>, результат якого передається відповідній змінній після завершення директиви parallel та інші.

Наведемо приклади, у яких застосовуються зазначені опції, та покажемо, як їх можна перетворити за допомогою паралельних конструкцій мови Ада. Спочатку розглянемо реалізацію опцій private та firstprivate:

```
with ada.text_io;
use  ada.text_io;
procedure omp_parallel_demo_2 is
  x, y : integer;
begin
  x := -1; y := -1;
  put_line("x=" & x'Img & " y=" & y'Img);
--omp parallel private(x), firstprivate(y)
--в паралельній області:
--присвоїти змінній x значення номеру (id) задачі,
--додати до змінної y значення номеру (id) задачі.
  put_line("x=" & x'Img & " y=" & y'Img);
--omp end parallel
  put_line("x=" & x'Img & " y=" & y'Img);
end omp_parallel_demo_2;
```

У послідовному варіанті програми, під час зміни значень змінних після директиви parallel, у процесі виведення на екран другий і третій раз вони будуть мати одні й ті ж самі значення. Після перетворення тексту програми, після зміни значень змінних у паралельних фрагментах, кожен із них повинен вивести власні значення зазначених змінних, а після паралельної області змінні повинні мати ті ж значення, що й до неї:

```
with ada.text_io;
use  ada.text_io;
with system.multiprocessors;
use  system.multiprocessors;
with manager_id;
use  manager_id;
procedure omp_parallel_demo_2 is
  x, y : integer;
begin
  x := -1; y := -1;
  put_line("x=" & x'Img & " y=" & y'Img);
--omp parallel private(x), firstprivate(y)
  omp_parallel_1: declare
    N_Threads : CPU := Number_Of_CPUs;
    id_man    : id_man_t;
    y0 : integer := y;
    task type t_par1;
    task body t_par1 is
      id : integer;
      x  : integer;
      y  : integer := y0;
    begin
      id_man.get_id(id);
      x := id;
      y := y + id;
      put("id =" & id'Img);
      put_line("x=" & x'Img & " y=" & y'Img);
    end t_par1;
    a_t_par1 : array (0..N_Threads-1) of t_par1;
  begin
    null;
  end omp_parallel_1;
--omp end parallel
  put_line("x=" & x'Img & " y=" & y'Img);
end omp_parallel_demo_2;
```

Як видно з тексту програми, для реалізації опції private, потрібно описати в тілі задачі копію змінної, що вказана в дужках. Для реалізації ж опції firstprivate, спочатку в блоці потрібно описати нову змінну, яку ініціювати значенням відповідної змінної до початку паралельного фрагменту програми, а потім при описі в тілі задачі локальної копії змінної, провести її ініціацію значенням щойно створеної нової змінної.

Можливі й інші варіанти забезпечення реалізації опції firstprivate. Наприклад, можна передати значення змінних до паралельного фрагменту в їхні локальні копії за допомогою механізму рандеву, це було зроблено в першому прикладі, під час передавання ідентифікаційного номера. Але цей спосіб потребує організації додаткових циклічних конструкцій і затримки в роботі паралельних фрагментів. Також у паралельному фрагменті можна перейменувати усі змінні, позначені як firstprivate. Тоді присвоєння цим змінним значень відповідників робиться при їх описі в тілі задачі без усіляких перешкод.

У наступному прикладі буде показана реалізація опцій shared та reduction:

```
with ada.text_io;
use  ada.text_io;
procedure omp_parallel_demo_3 is
  a, b : integer;
begin
  a := -1; b := -1;
  put_line("a=" & a'Img & " b=" & b'Img);
```

```
--omp parallel shared(a), reduction(+:b)
--в паралельній області:
--присвоїти змінній a і b значення номеру (id) задачі,
 put_line("a=" & a'Img & " b=" & b'Img);
--знайти суму усіх локальних копій змінної b.
--omp end parallel
 put_line("a=" & a'Img & " b=" & b'Img);
end omp_parallel_demo_3;
```

Так, як і в попередньому прикладі, у послідовному варіанті програми, при зміні значень змінних після директиви parallel, у процесі виведення на екран другий і третій раз вони будуть мати одні і ті ж самі значення. Після перетворення тексту програми, після зміни значень змінних у паралельних фрагментах, кожен з них повинен вивести власні значення зазначених змінних, а після паралельної області змінна, що позначена як shared, повинна дорівнювати одному з номерів, які були присвоєні задачам, а змінна з опції reduction – дорівнювати сумі значень усіх її локальних копій:

```
with ada.text_io;
use  ada.text_io;
with system.multiprocessors;
use  system.multiprocessors;
with manager_id;
use  manager_id;
procedure omp_parallel_demo_3 is
 a, b : integer;
pragma atomic(a);
pragma volatile(a);
begin
 a := -1; b := -1;
 put_line("a=" & a'Img & " b=" & b'Img);
--omp parallel shared(a), reduction(+:b)
 omp_parallel_1: declare
  N_Threads : CPU := Number_Of_CPUs;
  id_man    : id_man_t;
  protected red_man is
   procedure to_red(r_var : in integer);
   entry get_red(r_var : out integer);
  private
   m_r_var : integer;
   count   : natural := 0;
  end red_man;
  protected body red_man is
   procedure to_red(r_var : in integer) is
   begin
    if count = 0 then
     m_r_var := 0;
    end if;
    m_r_var := m_r_var + r_var;
    count := count + 1;
   end to_red;
   entry get_red(r_var : out integer)
    when count = integer(N_Threads) is
   begin
    r_var := m_r_var;
   end get_red;
  end red_man;
  task type t_par1;
  task body t_par1 is
   id : integer;
   b  : integer;
  begin
   id_man.get_id(id);
   a := id; b := id;
   red_man.to_red(b)
   put("id =" & id'Img);
   put_line("a=" & a'Img & " b=" & b'Img);
  end t_par1;
  a_t_par1 : array (0..N_Threads-1) of t_par1;
 begin
  red_man.get_red(b);
 end omp_parallel_1;
--omp end parallel
 put_line("a=" & a'Img & " b=" & b'Img);
end omp_parallel_demo_3;
```

З тексту програми видно, що для забезпечення опції shared, необхідно до відповідних змінних застосувати прагми (**pragma**) atomic та volatile, це забезпечить коректне колективне використання змінної. Щоб реалізувати дію опції reduction, у програмі використаний захищений модуль red_man, у межах якого в процедурі to_red відбувається додавання переданих у неї значень змінних та збільшення лічильника відпрацьованих паралельних ділянок (задач), а у вході get_red, після виконання бар'єрної умови, яка перевіряє чи усі задачі передали свої змінні для редукції, повертається результат додавання локальних копій змінної, що описана у відповідній опції. Описаний захищений модуль red_man, як і захищений модуль id_man, також можна винести в окремий пакет, який можна зробити налаштовуваним (generic) відносно типу змінної та операції редукції.

Редукцію можна реалізувати й за допомогою механізму рандеву, подібно до описаного вище способу передавання ідентифікаційного номера в задачу. Але недоцільність такого підходу, через використання додаткових циклів, є очевидною.

Із наведених прикладів цілком зрозуміло, що для реалізації директиви parallel засобами розпаралелювання мови Ада можна спроектувати компактний шаблон:

```
...
with system.multiprocessors;
use  system.multiprocessors;
with manager_id;
use  manager_id;
procedure omp_parallel_demo is
...
<використання виразів pragma atomic() та
 pragma volatile() для кожної shared змінної>
...
begin
...
--omp parallel private(<список>), firstprivate(<список>),
--          shared(<список>), reduction(+:<список>)
 omp_parallel_1: declare
  N_Threads : CPU := Number_Of_CPUs;
  id_man    : id_man_t;
  protected type red_man is
   procedure to_red(r_var : in integer);
   entry get_red(r_var : out integer);
  private
   m_r_var : integer;
   count   : natural := 0;
  end red_man;
  protected body red_man is
```

```
  ...
end red_man;
<опис дублікатів reduction змінних та ініціалізація
 їх відповідними змінними до паралельної області>
<створення екземпляру захищеного модуля red_men
 для кожної reduction змінної>
task type t_par1;
task body t_par1 is
  id : integer;
  <опис private змінних>
  <опис reduction змінних та ініціалізація їх
   відповідними дублікатами, описаними вище>
  ...
begin
  id_man.get_id(id);
  <основна послідовність операторів>
  <виклик процедури to_red(<reduction змінна>)
   для кожної reduction змінної>
end t_par1;
a_t_par1 : array (0..N_Threads-1) of t_par1;
begin
  <виклик процедури get_red(<reduction змінна>)
    для кожної reduction змінної>
 end omp_parallel_1;
--omp end parallel
...
```

5. Висновки

Отже, у статті представлена принципова можливість реалізації директив OpenMP у програмах на мові Ада. За зразком директиви parallel можна розробити представлення й усіх інших директив технології OpenMP, а також розробити програму автоматичної модифікації програмного коду з директивами в програмний код, що містить паралельні конструкції мови Ада. Для реалізації такої програми доцільно було б використати специфікацію семантичного інтерфейсу Ади (Ada Semantic Interface Specification – ASIS) [7].

Література

1. OpenMP Application Program Interface Version 3.1. July 2011. [Електронний документ]. Режим доступу: http://www.openmp.org/mp-documents/OpenMP3.1.pdf. Перевірено: 18.07.2012.
2. Антонов А.С. Параллельное программирование с использованием технологии OpenMP. – М.: Изд-во МГУ, 2009. – 76 с.
3. Малеванный М.С. Реализация директив OpenMP для языка PascalABC. NET. Магистерская диссертация. – Ростов-на-Дону, 2011. – 52 с.
4. Stpiczyński P. Ada as a language for programming clusters of SMPs. – Annales UMCS Informatica AI 1, 2003. – P.P. 73–79.
5. Ada Reference Manual ISO/IEC 8652:2007(E) Ed. 3. [Електронний документ]. Режим доступу: http://www.adapower.com/rm-95/index.html. Перевірено: 18.07.2012.
6. Гавва А. «Адское» программирование. Ada-95. Компилятор GNAT. [Електронний документ]. Режим доступу: http://ada-ru.org/V-0.4w/index.html. Перевірено: 18.07.2012.
7. ASIS-for-GNAT Reference Manual [Електронний документ]. Режим доступу: http://docs.adacore.com/asis-docs/asis_rm.html. Перевірено: 18.07.2012.

Abstract

Modern software is used in computers with several processing elements, therefore, it should be developed using the technology of parallel programming. Currently, there are several technologies for development of parallel programs. Special attention should be paid to the means that allow "to make parallel" developed consecutive program immediately by means of adding special instructions to its initial code; according to which the compilation environment will form parallel variant of the code and performed file. OpenMP technology belongs to the means of such type. Ada programming language has its own means for the development of parallel software, which allow developing a parallel program of any complexity. However, using the built-in parallelization of Ada language could complicate the process of software development and make the program text impossible for understanding and analysis. The means similar to OpenMp technology provide more convenience while paralleling algorithms. The opportunity to introduce OpenMp technology directives to Ada programming language is shown in this paper; the ways for replacement of directives by parallel language constructions are analyzed; the general consequence of actions is represented while transforming from consequent Ada program with OpenMp directives to parallel Ada program. The opportunities of implementing some OpenMP directives by means of Ada language are considered in the work. The variant of parallel directive and corresponding pattern are presented. The article includes the examples of program code

Keywords: OpenMP technology, program code conversion