УДК 681.3

ОСОБЛИВОСТІ КОМПІЛЯЦІЇ СХЕМ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

О. І. Захаров, С. Д. Погорілий.

Київський національний університет імені Тараса Шевченка. 01033, Київ, вул. Володимирська 64.
e-mail: sdp@rpd.univ.kiev.ua, zakharov@mail.univ.kiev.ua

Проаналізовано сигнатуру алгоритмічних алгебр Глушкова. З метою формалізованого запису паралельних алгоритмів сигнатуру доповнено операцією паралельної композиції операторів. Створено параметричний компілятор схем паралельних алгоритмів. В статті наведено приклад компіляції паралельного алгоритму.

The signature of Glushkov's algorithmic algebras is analyzed. For formal notation of parallel algorithms the operation of parallel composition is added to the signature. The parametric compiler of parallel algorithm schemes is developed. An example of parallel algorithm compilation is adduced in the article.

Однією з основних характеристик обчислювальної системи є такий показник, як продуктивність — величина, що показує, яку кількість арифметичних операцій він може виконати за одну секунду. Принципово важливими рішеннями в підвищенні продуктивності обчислювальних систем є:

- введення конвеєрної організації виконання процесорних команд;
- включення в систему команд векторних операцій, які дозволяють однією командою обробляти масиви даних;
- розподілення обчислень на множину процесорів.

Перші два напрямки підвищення продуктивності є апаратними, тому основна увага в збільшенні продуктивності програмними засобами приділяється методам паралельного обчислення для багатопроцесорних систем.

З іншого боку не менш актуальним напрямком дослідження та оптимізації обчислювальних алгоритмів є формалізований синтез програм, суть якого полягає у створенні схеми алгоритму та подальшої трансформації її у програму шляхом підстановки замість формальних специфікацій їх реалізацій певною мовою програмування. В [1] розглянуто інструментальний комплекс автоматизованого синтезу програм у візуальних середовищах програмування, в якому схеми алгоритмів представлені мовою САА/1, що базується на системі алгоритмічних алгебр В. М. Глушкова (САА) [2, 3].

Наявність систем формалізованого синтезу програм [1, 4] та розвиток методів розподілення обчислень між процесорами висуває дві задачі:

- доповнення САА операціями паралельного виконання операторів;
- створення компілятора схем паралельних алгоритмів.

Розширення сигнатури САА

Проаналізуємо існуючі операції САА $\tilde{O}_{SAA}$ з точки зору паралельного виконання.

Булеві операції — кон'юнкція ($\alpha_1 \wedge \alpha_2$), диз'юнкція ($\alpha_1 \vee \alpha_2$), заперечення ($\bar{\alpha}$)

Паралельне обчислення булевих функцій неефективне, тому що майже всі сучасні процесори виконують їх за одну машинну команду, і навіть за один процесорний такт. Так як синхронізація процесів вимагає значної кількості машинного коду, то час паралельного обчислення булевого виразу значно перевищує час виконання на одному процесорі.

Композиція ($O_1 * O_2$)

Нехай $O_1 = O_1(v_1^1, v_1^2, ..., v_1^{n_1}, u_1^1, u_1^2, ..., u_1^{k_1})$, $O_2 = O_2(v_2^1, v_2^2, ..., v_2^{n_2}, u_2^1, u_2^2, ..., u_2^{k_2})$, де $v_1^i, v_2^j, i = \overline{1...n_1}, j = \overline{1...n_2}$ — змінні, значення яких використовується у відповідних операторах; $u_1^i, u_2^j, i = \overline{1...k_1}, j = \overline{1...k_2}$ — змінні, значення яких модифікується. Операцію композиції можна виконати паралельно лише за умови $\{v_1^i\} \cap \{u_2^j\} = \varnothing, i = \overline{1...n_1}, j = \overline{1...k_2}$ та $\{v_2^i\} \cap \{u_1^j\} = \varnothing, i = \overline{1...n_2}, j = \overline{1...k_1}$, тобто тоді і тільки тоді, коли вона має властивість комутативності $O_1 * O_2 \equiv O_2 * O_1$ (будемо казати, що оператори O_1 та O_2 незалежні). Отже, введемо операцію паралельного виконання операторів O_1 та O_2: $O_1 \tilde{*} O_2$. Зрозуміло, що наведена операція також має властивість комутативності $O_1 \tilde{*} O_2 \equiv O_2 \tilde{*} O_1$.

Альтернатива (α-диз'юнкція) $([\alpha]O_1, O_2)$

При виконанні операції паралельно можна обчислювати як умову α, так і оператори O_1 та O_2. Слід зауважити, що після визначення значення умови α необхідно виконати лише один оператор O_1 (умова істинна) чи O_2 (умова хибна). Паралельне обчислення операторів O_1 і O_2 можливе лише за умови, коли α і O_1, α і O_2

є попарно незалежними, та є ефективним, якщо час обчислення α, O_1 і O_2 збігається за порядком. В цьому випадку існує перетворення $([\alpha]O_1, O_2) = O'_\alpha * ([\alpha']O_1, O_2)$ таке, що час виконання умови α' буде набагато меншим за α (α' зведеться до обчислення булевих функцій), а, отже, паралельне виконання альтернативи зведеться до паралельної композиції $O'_\alpha \tilde{*} ([\alpha']O_1, O_2)$.

Цикл (α-ітерація) $\{[\alpha]O\}$

В загальному випадку цикл неможливо виконувати паралельно, але якщо обчислення довільної ітерації не залежить від попередніх, або може бути зведене певним перетворенням до незалежного, то цикл можна розбити на декілька $\{[\alpha]O\} = \{[\alpha_1]O_1\} * \{[\alpha_2]O_2\} * ... * \{[\alpha_n]O_n\}$ (кількість n дорівнює кількості процесів) так, що кожний утворений цикл може виконуватись в окремому процесі. Прикладом може бути обчислення суми ряду, кожний член якого не залежить від попередніх, наприклад $\sum_{i=1}^{n} \frac{1}{i}$. Отже знову бачимо, що паралельне обчислення циклу зводиться до паралельного виконання композицій $\{[\alpha]O\} = \{[\alpha_1]O_1\} \tilde{*} \{[\alpha_2]O_2\} \tilde{*} ... \tilde{*} \{[\alpha_n]O_n\}$.

З вищенаведеного видно, що для формального запису паралельного алгоритму в сигнатурі алгоритмічної алгебри Глушкова достатньо ввести одну операцію паралельного виконання операторів $O_1 \tilde{*} O_2$.

Введення узагальнених операторів синхронізації процесів

Очевидно, що кожний з процесів, які виконують паралельний алгоритм, виконує певний машинний код, який може бути представлений деяким алгоритмом, записаним у сигнатурі САА. Тоді зрозуміло, що задача компіляції паралельної схеми зводиться до її тотожного перетворення у кілька однопроцесорних формул, кількість яких дорівнює кількості виконуючих процесів (цей параметр повинен бути заданий на етапі компіляції). Отже, до операції паралельного виконання операторів $O_1 \tilde{*} O_2$ необхідно застосувати тотожне перетворення, в результаті якого отримаємо дві формули: $O_1^b * O_1 * O_1^e$ та $O_2^b * O_2 * O_2^e$, де O_1^b та O_2^b — початкові, а O_1^e та O_2^e — кінцеві синхронізуючі оператори.

Конкретне представлення цих операторів залежить насамперед від:

- архітектури розподіленої системи,
- операційної системи,
- мови програмування.

Розглянемо види синхронізації процесів та визначимо загальне представлення синхронізуючих операторів, конкретна реалізація яких буде залежати від факторів, наведених вище.

Синхронізація за часом виконання

Виникають ситуації, при яких деякий процес для продовження обчислень повинен чекати завершення виконання певних операторів, які в даний момент часу обробляються іншими процесами. Відповідно процеси, які закінчили обробляти деякий оператор, повинні про це повідомити. Тому доцільно ввести оператори очікування та повідомлення, а також булеву функцію перевірки стану певного оператору:

Таблиця 1. Оператори синхронізації за часом виконання.

$WaitAny(O_1, O_2, ..., O_n)$	Блокуючий оператор (виконання завершиться тільки при виконанні певної умови) очікування завершення хоча б одного з операторів $O_1, O_2, ..., O_n$
$WaitAll(O_1, O_2, ..., O_n)$	Блокуючий оператор очікування завершення всіх операторів $O_1, O_2, ..., O_n$.
$Test(O)$	Булева функція перевірки стану оператора O, яка буде істинною якщо O виконаний, в протилежному випадку хибною
$Signal(O)$	Сигнал про завершення процесом оператора O

Синхронізація даних

У випадку системи з загальною пам'яттю, в якій дані доступні одночасно всім процесам, не потрібно виконувати синхронізацію змінних. В системах з розподіленою пам'яттю кожний процес використовує власну область пам'яті, яка недоступна для інших процесів, то виникає необхідність у синхронізації даних, які отримуються в результаті обчислень. В цьому випадку єдиним можливим механізмом взаємодії між процесами є механізм передачі повідомлень. Звичайно, для досягнення максимальної продуктивності реалізація механізму повідомлень повинна базуватися на властивостях архітектури багатопроцесорної системи. Але в більшості випадків синхронізацію можна виконати найбільш загальними операторами обміну змінними між процесами. Слід зауважити, що необхідно ввести як повідомлення між двома процесами, так і широкомовні повідомлення (broadcast). Отже визначимо наступні оператори:

Таблиця 2. Оператори синхронізації за даними.

$Send(P, v_1, v_2, ..., v_n)$	Передача процесу P значень змінних $v_1, v_2, ..., v_n$.
$SendBroadcast(v_1, v_2, ..., v_n)$	Широкомовне повідомлення з передачею значень змінних $v_1, v_2, ..., v_n$
$Receive(P, v_1, v_2, ..., v_n)$	Одержання з процесу P значень змінних $v_1, v_2, ..., v_n$.

Необхідно також визначити оператори початкової ініціалізації та звільнення ресурсів:

Таблиця 3. Оператори початкової ініціалізації та звільнення ресурсів.

Initialize	Оператор початкової ініціалізації. Повинен бути першим оператором процесу
Finalize	Оператор звільнення ресурсів, які використовувались для забезпечення синхронізації та обміну повідомленнями між процесами. Повинен бути останнім

Розглянемо приклад застосування вищенаведених операторів. Нехай маємо оператор $O = O_1(a^{rw}, b^r) \tilde{*} O_2(b^r, c^{rw}) \tilde{*} O_3(a^{rw}, c^{rw})$, який представлено композицією деяких простих операторів O_1, O_2, O_3, та змінними a, b, c (верхній індекс змінної r означає, що її значення використовується при виконанні даного оператору, w — значення змінної модифікується). Для системи з двома процесорами P_1 та P_2 відповідно отримаємо:

$P_1 : Initialize * Send(P_2, a, b) * O_2 * WaitAny(O_1) * Recieve(P_2, a) * O_3 * Finalize$

$P_2 : Initialize * Receive(P_1, a, b) * O_1 * Signal(O_1) * Send(P_1, a) * Finalize$

На наведених нижче рисунках зображено виконання оператору O у випадку однопроцесорної системи (рис 1а) та двохпроцесорної системи (рис 1б).

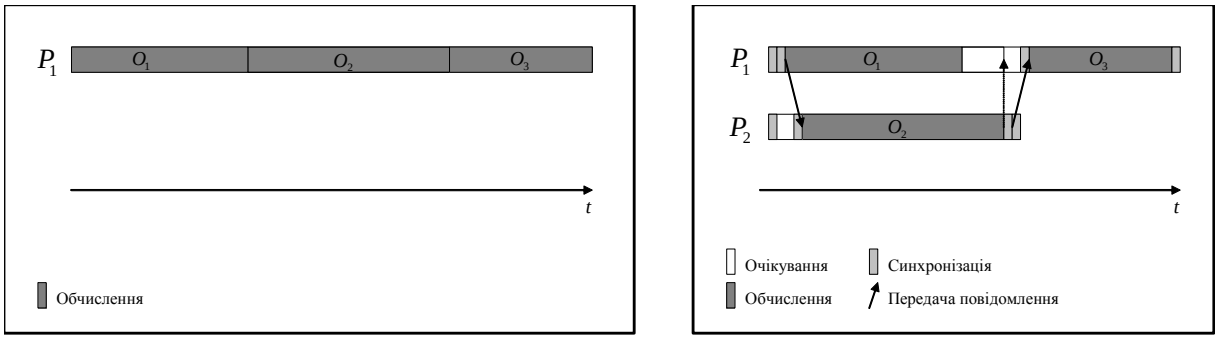

Рис 1. Виконання оператору O: а) на однопроцесорній системі; б) на двохпроцесорній системі.

Компілятор схем паралельних алгоритмів

На рис. 2 наведено структурну схему компілятора

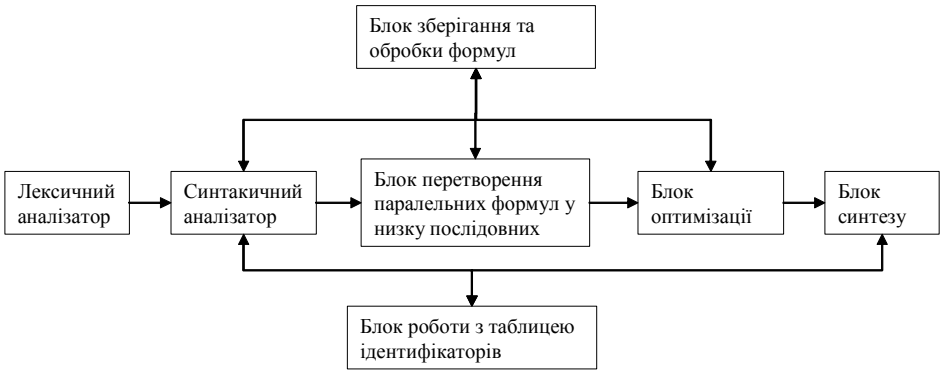

Рис 2. Структурна схема компілятора

Блок лексичного аналізу виконано у вигляді скінченого детермінованого автомату. Так як в мові САА/1 тип лексеми однозначно визначається в тому місці тексту, де вона знаходиться, то застосовано прямий

лексичний аналізатор. Для запису паралельних алгоритмів мова САА/1 доповнена лексемою паралельного виконання операторів "||".

На вхід **синтаксичного аналізатору** подається потік лексем, який надходить з блоку лексичного аналізу. Так як алгоритм можна представити у вигляді графу, то на виході синтаксичного аналізатору будується дерево, що відповідає алгоритму вхідної програми. Якщо ідентифікатор головної формули описується окремою формулою, то дерево, яке їй відповідає, підставляється замість цього ідентифікатора у головну формулу. Таким чином, після завершення синтаксичного аналізу будується дерево, листками якого є елементарні ідентифікатори, які не задаються формулами. Для таких ідентифікаторів необхідна наявність реалізації в базі даних.

Блок зберігання та обробки формул необхідний для виконання основних операцій над деревами формул, такі як вставка піддерева, обхід дерева, тощо.

Модуль роботи з таблицею ідентифікаторів – єдиний модуль компілятора, що безпосередньо працює з базою даних. Після проведення синтаксичного аналізу та перетворень формули алгоритму блок вилучає з бази даних необхідну інформацію (реалізації, опис змінних, тощо) для елементарних ідентифікаторів.

Блок перетворення паралельних формул у низку послідовних необхідний для компіляції паралельних алгоритмів. Його робота полягає у пошуку в дереві алгоритму операцій паралельної композиції, та тотожному перетворенні дерева у декілька, які представляють послідовні алгоритми з включенням необхідних операторів синхронізації.

Важливим етапом роботи компілятора є тотожні перетворення формул з метою оптимізації алгоритму та вихідної програми, які виконує **блок оптимізації**. Модуль виконує наступні типи оптимізації:

- за часом виконання алгоритму,
- за розміром коду вихідної програми,
- за розміром сегменту даних, тобто розміром глобальних змінних,
- за розміром стеку програми, тобто розміром локальних змінних процедур.

Останній блок компілятора – **блок синтезу коду**. Він служить для генерації тексту вихідної програми на основі побудованих формул алгоритму. Система може настроюватись на різні вихідні мови програмування. Таке настроювання здійснюється шляхом застосування технології підключення додаткових модулів (plug-ins).

Приклад

Розглянемо приклад паралельного обчислення на двохпроцесорній системі числа π за формулою

$$\frac{\pi}{4} = \sum_{i=1}^{2n} (-1)^{i+1} \frac{1}{2i-1} = \sum_{i=1}^{n} \frac{1}{4i-3} - \sum_{i=1}^{n} \frac{1}{4i-1}$$

САА формула для цього прикладу має вигляд:

$$"\text{Cycle1 initialization}" = "\text{Index1 assign to 1}" * "\text{Sum1 assign to 0}"$$

$$"\text{Cycle1 body}" = "\text{Add to sum1 next member}" * "\text{Increment index1}"$$

$$"\text{Cycle1}" = "\text{Cycle1 initialization}" * \{[" \text{Cycle1 condition}"] "\text{Cycle1 body}"\}$$

$$"\text{Cycle2 initialization}" = "\text{Index2 assign to 1}" * "\text{Sum2 assign to 0}"$$

$$"\text{Cycle2 body}" = "\text{Add to sum2 next member}" * "\text{Increment index2}"$$

$$"\text{Cycle2}" = "\text{Cycle2 initialization}" * \{[" \text{Cycle2 condition}"] "\text{Cycle2 body}"\}$$

$$"\text{Pi Calculation}" = "\text{Cycle1}" \tilde{*} "\text{Cycle2}" * "\text{Calculate sum}" * "\text{Show answer}"$$

Та ж схема, записана мовою САА/1 виглядає так:

```
scheme “Pi Calculation” = “CalcPi”;
“Pi Calculation” = “Cycle1” || “Cycle2” * “Calculate sum” * “Show answer”;
“Cycle1” = “Cycle1 initialization” * while ‘Cycle1 condition’ do (“Cycle1 body”);
“Cycle2” = “Cycle2 initialization” * while ‘Cycle2 condition’ do (“Cycle2 body”);
“Cycle1 initialization” = “Index1 assign to 1” * “Sum1 assign to 0”;
“Cycle1 body” = “Add to sum1 next member” * “Increment index1”;
“Cycle2 initialization” = “Index2 assign to 1” * “Sum2 assign to 0”;
“Cycle2 body” = “Add to sum2 next member” * “Add to sum2 next member”;
end.
```

Список понять алгоритму, їх реалізації та параметри

Таблиця 4. Поняття алгоритму, їх реалізації та параметри.

Поняття	Реалізація	Оператор чи умова	Змінні				
			Назва	Тип	Область видимості	Читання	Запис
Index1 assign to 1	I1:=1	О	I1	Integer	Параметр		•
Sum1 assign to 0	S1:=0	О	S1	Extended	Глобальна		•
Add to sum1 next member	S1:=S1+1/(4*I1-3)	О	S1	Extended	Глобальна	•	•
			I1	Integer	Параметр	•	
Increment index1	Inc(I1)	О	I1	Integer	Параметр	•	•
Cycle1 condition	I1<=N	У	I1	Integer	Параметр	•	
Index2 assign to 1	I2:=1	О	I2	Integer	Параметр		•
Sum2 assign to 0	S2:=0	О	S2	Extended	Глобальна		•
Add to sum2 next member	S2:=S2+1/(4*I2-1)	О	S2	Extended	Глобальна	•	•
			I2	Integer	Параметр	•	
Increment index2	Inc(I2)	О	I2	Integer	Параметр	•	•
Cycle2 condition	I2<=N	У	I2	Integer	Параметр	•	
Calculate sum	S=4*(S1-S2)	О	S	Extended	Глобальна		•
			S1	Extended	Глобальна	•	
			S2	Extended	Глобальна	•	
Show answer	WriteLn(S)	О	S	Extended	Глобальна	•	

Синтезована програма

```
program CalcPi;

{$APPTYPE CONSOLE}

uses Windows;

const
  N = 1000000;

var
  S, S1, S2: Extended;
  I1: Integer;
  Process1Id: Longword;
  Process1Handle: THandle;
  Signal1Handle: THandle;

function Process1(lpParameter: Pointer): Longword; stdcall;
var
  I2: Integer;
begin
  S2 := 0;
  I2 := 1;
  while (I2 < N) do
  begin
    S2 := S2 + 1 / (4 * I2 - 1);
    Inc(I2);
  end;
  SetEvent(Signal1Handle);
  ExitThread(0);
  Result := 0;
end;

begin
  Signal1Handle := CreateEvent(nil, False, False, nil);
```

```
  Process1Handle := CreateThread(nil, 0, @Process1, nil, 0, Process1Id);
  S1 := 0;
  I1 := 1;
  while (I1 <= N) do
  begin
    S1 := S1 + 1 / (4 * I1 - 3);
    Inc(I1);
  end;
  WaitForSingleObject(Signal1Handle, INFINITE);
  s := 4 * (S1 - S2);
  WriteLn(S);
  WaitForSingleObject(Process1Handle, INFINITE);
  CloseHandle(Process1Handle);
  CloseHandle(Signal1Handle);
end.
```

Синтез виконаний для операційної системи Microsoft Windows NT та мови програмування Borland Delphi.

Висновки

В результаті виконання роботи:

- Виконано аналіз сигнатури САА і запропоновано її доповнення операцією паралельного виконання операторів та засобами синхронізації за часом виконання та за даними.
- Створено параметричний компілятор для трансформування схем паралельних алгоритмів у програми для багатопроцесорних систем. Компілятор реалізовано у середовищі Borland Delphi 7.0. Перевагою створеного компілятора є можливість параметричного настроювання на різні цільові мови програмування.

Література

1. *Погорілий С.Д., Захаров О.І.* Створення інструментальних засобів синтезу програм у візуальних середовищах. // Проблемы программирования // Специальный выпуск №1-2, 2002.
2. *Погорілий С.Д.* Автоматизація наукових досліджень. Основоположні теоретичні відомості. Програмне забезпечення. За редакцією академіка АПН України Третяка О.В. — К: Видавничо-поліграфічний центр “Київський університет”, 2002. — 288 с.
3. *Цейтлин Г.Е.* Введение в алгоритмику. — Киев: Сфера, 1998. — 310 с.
4. *Ющенко Е.Л., Цейтлин Г.Е., Грицай В.П., Терзян Т.К.* Многоуровневое структурное проектирование программ: Теоретические основы, инструментарий. — Москва: Финансы и статистика, 1989. — 208 с.
5. Математическая энциклопедия. — Москва: Советская энциклопедия, 1985.
6. *Воеводин В. В., Воеводин Вл. В.* Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 600 с.
7. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum. — Version 1.1. 1995. — http://www-unix.mcs.anl.gov/mpi
8. MPI: The Complete Reference. — http://rsusu1.rnd.runnet.ru/ncube/mpi/mpibook/mpi-book.html