МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ФИЗИКИ

КАФЕДРА РАДИОАСТРАНОМИИ

Лабораторная работа на тему:

**Высокопроизводительные вычислительные системы**

Выполнил студент группы 06-722

Миндугулов Мансур

г.Казань

2020

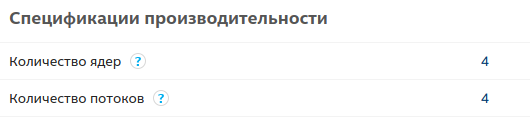
**Цель работы:**  провести эксперимент по анализу ускорения вычислений на кластере.

**Ход работы:**

**Работа проводится на OS Linux mint 19.3 Cinnamon**

**Процессор Intel© Core™ i5-7300HQ CPU @ 2.50GHz × 4**

Из характеристик с официального сайта intel.com, узнали, что данный процессор имеет 4 физических ядра по 1 логическому ядру в каждом.



Значит в нашей задаче будем использовать от 1 до 4 потоков при параллельных вычислениях.

Поставленная перед нами задача формулируется следующим образом:

**Вычислить матрицу C, как функцию от заданнх матриц**

**С = A2 + B\*8 + 3**

Составили и отладили программу, решающую задачу, на языке С++ в последовательном и параллельном режиме вычислений.

1. #include <iostream>
2. #include <omp.h>
3. #include <cstdlib>
4. #include <time.h>
5. #include <stdio.h>
7. **using** **namespace** std;
9. // функция генерации случайного числа из диапазона от range\_min до range\_max включительно
10. **int** rrand(**int** range\_min, **int** range\_max) {
11. **return** rand() % (range\_max - range\_min + 1) + range\_min;
12. }
14. **const** **long** **long** **int** ARRSIZE = 1500;
15. **const** **int** ABSLIMIT = 9;
16. **int** A[ARRSIZE][ARRSIZE];
17. **int** B[ARRSIZE][ARRSIZE];
18. **int** AA[ARRSIZE][ARRSIZE];
19. **int** B8[ARRSIZE][ARRSIZE];
20. **int** RESULT[ARRSIZE][ARRSIZE];
22. **void**  print\_matrix(**int** A[][ARRSIZE])
23. {
24. **int** i = 0;
25. **int** j = 0;
26. **while** (i < ARRSIZE)
27. {
28. **while** (j < ARRSIZE)
29. {
30. //   printf("%-5d ", A[i][j]);
31. j++;
32. }
33. j = 0;
34. i++;
35. // printf("\n");
36. }
37. }
39. **int** main(**void**)
40. {
42. // заполнили две матрицы A, B рандомными числами
43. **int** i = 0;
44. **int** j = 0;
45. **int** k = 0;
46. **int** nthreads;
47. clock\_t clock\_timer;
48. **double** wall\_timer;
50. **while** (i < ARRSIZE)
51. {
52. **while** (j < ARRSIZE)
53. {
54. A[i][j] = rrand(-ABSLIMIT, ABSLIMIT);
55. B[i][j] = rrand(-ABSLIMIT, ABSLIMIT);
56. j++;
57. }
58. j = 0;
59. i++;
60. }
61. // запускаем в расспараллеливание операцию А\*A
62. **for** (nthreads = 1; nthreads <=4; ++nthreads) {
63. clock\_timer = clock();
64. wall\_timer = omp\_get\_wtime();
65. **for** (i = 0; i < ARRSIZE; i++)
66. {
67. **for** (j = 0; j < ARRSIZE; j++)
68. {
69. AA[i][j] = 0;
70. #pragma omp parallel for num\_threads(nthreads)  reduction(+:AA[i][j])
71. **for** (k = 0; k < ARRSIZE; k++)
72. {
73. AA[i][j] += A[i][k] \* A[k][i];
74. }
75. }
76. }
77. // запускаем в расспараллеливание операцию В\*8
78. #pragma omp parallel for num\_threads(nthreads)
79. **for** (i = 0; i < ARRSIZE; i++)
80. {
81. **for** (j = 0; j < ARRSIZE; j++)
82. {
83. B8[i][j] = B[i][j] \* 8;
84. }
85. }
86. // запускаем в расспараллеливание операцию AA+B8+3
87. #pragma omp parallel for num\_threads(nthreads)
88. **for** (i = 0; i < ARRSIZE; i++)
89. {
90. **for** (j = 0; j < ARRSIZE; j++)
91. {
92. RESULT[i][j] = AA[i][j] + B8[i][j] + 3;
93. }
94. }
95. std::cout << "Использовано потоков: " << nthreads
96. << ". Время выполнения вычислений: " <<  omp\_get\_wtime() - wall\_timer << " сек" << "\n";
97. }
98. // printf("Result of calculates is\n");
99. //  print\_matrix(RESULT);
101. **return** 0;
102. }

Для распараллеливания была выбрана библиотека openMP.

Для этого в шапку программы добавили строку

1. #include <omp.h>

Очевидно, что цикл for можно распараллелить и обрабатывать сразу несколькими ядрами процессора, поскольку вычисление значения любого элемента c[k] никак не зависит от остальных элементов массива c. В листинге 1 показано, как можно это сделать с помощью OpenMP на примере подсчета квадрата матрицы А.

Листинг 1.

1. **for** (i = 0; i < ARRSIZE; i++)
2. {
3. **for** (j = 0; j < ARRSIZE; j++)
4. {
5. AA[i][j] = 0;
6. #pragma omp parallel for num\_threads(nthreads)
7. **for** (k = 0; k < ARRSIZE; k++)
8. {
9. AA[i][j] += A[i][k] \* A[k][i];
10. }
11. }
12. }

Количеством потоков можно легко управлять с помощью прагмы с аргументом num\_threads. Ниже представлен код из листинга 2 с заданным количеством потоков (int nthreads потоков) на примере подсчета умножения матрицы В на число:

Листинг 2.

1. // запускаем в расспараллеливание операцию В\*8
2. #pragma omp parallel for num\_threads(nthreads)
3. **for** (i = 0; i < ARRSIZE; i++)
4. {
5. **for** (j = 0; j < ARRSIZE; j++)
6. {
7. B8[i][j] = B[i][j] \* 8;
8. }
9. }

**Директива reduction**

Директива OpenMP reduction позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков последующего параллельного структурного блока. В нашей программе в каждом параллельном потоке определена локальная переменная AA[i][j] для вычисления частичных сумм каждого индекса перемножения. После завершения параллельных потоков все локальные переменные AA[i][j] суммируются, а результат сохраняется в одноименной общей (глобальной) переменной AA[i][j].

В листинге 3 показано, как это реализовано в нашей программе с помощью OpenMP

Листинг 3

1. **for** (i = 0; i < ARRSIZE; i++)
2. {
3. **for** (j = 0; j < ARRSIZE; j++)
4. {
5. AA[i][j] = 0;
6. #pragma omp parallel for num\_threads(nthreads)  reduction(+:AA[i][j])
7. **for** (k = 0; k < ARRSIZE; k++)
8. {
9. AA[i][j] += A[i][k] \* A[k][i];
10. }
11. }
12. }

Для того чтобы значительно увеличить временную сложность вычисления функций мы решили использовать матрицы размером 1500х1500 элементов

Размеры массивов передаем в функцию глобальной переменной ARRSIZE (см. листинг 4)

Листинг 4

1. **const** **long** **long** **int** ARRSIZE = 1500;

Прагма parallel for помогает распределить рабочую нагрузку цикла for между несколькими потоками, каждый из которых может обрабатываться отдельным ядром процессора; таким образом общее время вычислений существенно снижается. Это подтверждается в листинге 5.

Листинг 5. Пример с использованием API-функции omp\_get\_wtime

1. clock\_t clock\_timer;
2. **double** wall\_timer;
4. **while** (i < ARRSIZE)
5. {
6. **while** (j < ARRSIZE)
7. {
8. A[i][j] = rrand(-ABSLIMIT, ABSLIMIT);
9. B[i][j] = rrand(-ABSLIMIT, ABSLIMIT);
10. j++;
11. }
12. j = 0;
13. i++;
14. }
15. // запускаем в расспараллеливание операцию А\*A
16. **for** (nthreads = 1; nthreads <=4; ++nthreads) {
17. clock\_timer = clock();
18. wall\_timer = omp\_get\_wtime();
19. **for** (i = 0; i < ARRSIZE; i++)
20. {
21. **for** (j = 0; j < ARRSIZE; j++)
22. {
23. AA[i][j] = 0;
24. #pragma omp parallel for num\_threads(nthreads)  reduction(+:AA[i][j])
25. **for** (k = 0; k < ARRSIZE; k++)
26. {
27. AA[i][j] += A[i][k] \* A[k][i];
28. }
29. }
30. }
31. // запускаем в расспараллеливание операцию В\*8
32. #pragma omp parallel for num\_threads(nthreads)
33. **for** (i = 0; i < ARRSIZE; i++)
34. {
35. **for** (j = 0; j < ARRSIZE; j++)
36. {
37. B8[i][j] = B[i][j] \* 8;
38. }
39. }
40. // запускаем в расспараллеливание операцию AA+B8+3
41. #pragma omp parallel for num\_threads(nthreads)
42. **for** (i = 0; i < ARRSIZE; i++)
43. {
44. **for** (j = 0; j < ARRSIZE; j++)
45. {
46. RESULT[i][j] = AA[i][j] + B8[i][j] + 3;
47. }
48. }
49. std::cout << "Использовано потоков: " << nthreads
50. << ". Время выполнения вычислений: " <<  omp\_get\_wtime() - wall\_timer << " сек" << "\n";

В листинге 5 мы измеряем время выполнения шести циклов for, увеличивая при этом количество потоков. API-функция omp\_get\_wtime возвращает затраченное фактическое время (в секундах), прошедшее с начала заданной точки отсчета. Таким образом, значение omp\_get\_wtime() - wall\_timer возвращает фактическое время выполнения цикла for. Системный вызов clock() используется для оценки времени, затраченного центральным процессором на выполнение всей программы, т. е. прежде чем получить итоговый результат, мы суммируем все эти временные интервалы с учетом потоков.

Во время вычислений мы делали скриншоты использования ЦП, чтобы наглядно подвердить наши действия.

На рис.1 приведён скриншот выполнения вычислений и график того, как использовались системные ресурсы во время вычислений. По графику использования ЦП хорошо видно, как с увеличением числа потоков, последовательно включались в работу ядра процессора. На моем компьютере с процессором Intel Core i5 я получил результаты, приведенные на рисунке 1

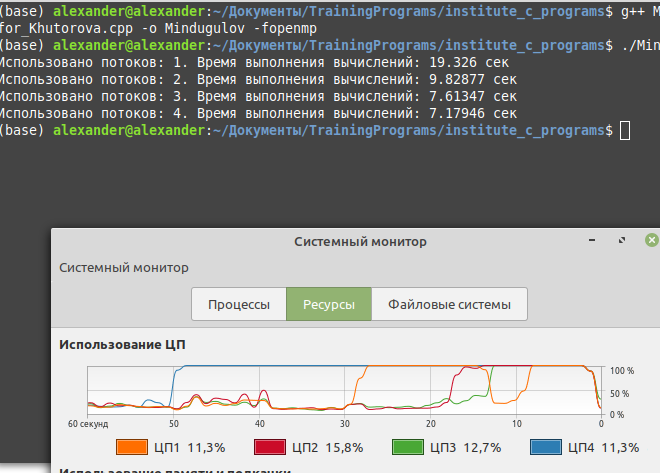


Рис. 1

**ВЫВОД:** Время выполнения программы постоянно уменьшалось при увеличении количества потоков, которые выполнялись параллельно отдельными процессорными ядрами. **Мы экспериментально продемонстрировали, что при увеличении потоков можно добиться существенного уменьшения времени выполнения программы. В нашем случае, увеличение числа потоков до 4х дало прибавку в скорости в 62.85% (12.14654 сек). Мы более чем в два раза выиграли в скорости при использовании многопоточности, по сравнению с однопоточным режимом вычислений! Также мы наглядно продемонстировали на временном графике использования ЦП, что при увеличении числа потоков у нас действительно включались в работу новые ядра процессора.**