Лабораторная работа №2

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Последовательную программу из лабораторной работы 1, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP. Реализовать два варианта программы:

• Вариант 1: для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for,

• Вариант 2: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.

Уделить внимание тому, чтобы при запуске программы на различном числе OpenMP-потоков решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

1. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: от 1 до числа доступных в узле. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
2. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков. 4. На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы

Реализация алгоритма первой лабораторной работы была переписана в двух вариантах с использованием OpenMP: в первом для каждого распараллеливаемого цикла создавалась отдельная секция #pragma omp parallel for, во втором же варианте одна секция pragma охватывает весь алгоритм.

Измерение времени работы программ (n = 11000):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Вариант 1 | 51,082 | 26,325 | 19,388 | 14,929 | 21,922 | 18,62 | 16,59 | 14,603 |
| Вариант 2 | 51,127 | 26,311 | 19,192 | 14,751 | 21,472 | 18,474 | 16,119 | 14,55 |

Ускорение:

Ускорением параллельного алгоритма называют отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Вариант 1 | 1,000 | 1,940 | 2,635 | 3,422 | 2,330 | 2,743 | 3,079 | 3,498 |
| Вариант 2 | 1,000 | 1,943 | 2,664 | 3,466 | 2,381 | 2,768 | 3,172 | 3,514 |

Эффективность:

Эффективностью работы программы называется отношение ускорения программы к количеству используемых потоков.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Вариант 1 | 100% | 97% | 88% | 86% | 47% | 46% | 44% | 44% |
| Вариант 2 | 100% | 97% | 89% | 87% | 48% | 46% | 45% | 44% |

Как можно видеть на полученных графиках эффективность работы программы падает с увеличением количества потоков. Это обусловлено увеличением накладных расходов. При использовании большего количества потоков увеличивается количество операций по управлению потоками, синхронизации и распределению задач между потоками.

Также виден скачок во времени выполнения программы между 4 и 5 потоками. Скорее всего это связано с тем, что на моем компьютере 5-ый и все последующие потоки являются виртуальными. Т.е. на 4 физических ядрах располагается по 2 потока на ядро. И при использовании виртуальных потоков, скорее всего, увеличивается время, требуемое на синхронизацию потоков и прочие операции с потоками.

В сравнении между 1 и 2 вариантом распараллеливания все графики накладываются друг на друга, что, фактически, говорит о том, что «разрезание» всего алгоритма и «разрезание» отдельных циклов одинаково эффективны и имеют лишь разную структуру параллельных вычислений.

Директива **schedule** определяет, каким образом итерации цикла делятся между потоками группы.

Если определена директива **schedule(static, x)** , то итерации делятся на порции длинной, равной **x**. Порции статически назначаются потокам в группе циклическим образом в порядке их номеров. Если размер порции определён, то всё количество итераций делится на порции, которые приблизительно равны между собой, и каждому потоку назначается одна порция. Если **x** не указать явно, то он по умолчанию равен 1. Таким образом каждый поток выполняет N/p итераций, где N- количество итераций цикла, p - число потоков.

Если определена директива **schedule (dynamic, x)**, то порции итераций длиной **x** назначаются каждому потоку. Когда поток завершит присвоенную ей порцию итераций, ей динамически назначается другая порция, пока все порции не закончатся. Следует заметить, что последняя назначаемая порция может иметь меньшее число итераций.

Если директива **schedule(guided, x)** определена, то итерации назначаются потокам порциями уменьшающихся размеров. Когда поток завершает выполнение назначенной ей порции итераций, то динамически назначается другая порция до тех пор, пока их не останется. Если значение **x** равно 1, то длина порции примерна равна частному от деления числа не назначенных итераций на число потоков. Длина порции уменьшается показательным образом до 1. Если **x** равен k (k > 1), то размеры порций уменьшаются показательно до k, за исключением того, что последняя порция может иметь меньше чем k итераций.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1 вариант** | | | **2 вариант** | | |
| **static** | **dynamic** | **guided** | **static** | **dynamic** | **guided** |
| **1** | 14,832 | 40,904 | 15,476 | 14,809 | 27,824 | 14,778 |
| **100** | 15,015 | 15,137 | 15,098 | 15,061 | 14,991 | 14,868 |
| **500** | 16,011 | 16,128 | 16,223 | 15,965 | 15,95 | 16,099 |
| **1000** | 16,025 | 15,939 | 16,748 | 16,107 | 16,01 | 16,534 |
| **2000** | 20,815 | 20,686 | 20,751 | 20,522 | 20,444 | 20,543 |
| **2750** | 14,782 | 14,745 | 15,028 | 15,02 | 14,743 | 14,894 |

Приложение 1. 1ый вариант

1. #**include** <stdio.h>
2. #**include** <stdlib.h>
3. #**include** <math.h>
4. #**include** <time.h>
5. #**include** <omp.h>
6. double\*\* **createMatrix** (const int n);
7. void **deleteMatrix** (double\*\* matrix, const int m);
8. double\* **createSinVector** (const int n, const double t);
9. double\* **createVector** (const double val, const int maxM);
10. void **deleteVector** (double\* vector);
11. void **mulMatrixVector** (double \*\*matrix, double \*vector, double \*res, const int n);
12. void **sub** (double\* mainVector, const double\* vector, const int n);
13. void **mulByScalar** (double\* vector, double scalar, int n);
14. double **updateX** (double\*\* matrix, double\* bVector, double\* xVector, double\* yVector, double\* AyVector, const int n);
15. void **setVectorZero** (double\* vector, int n);
16. double **scalarMul** (const double\* vector1, const double\* vector2, const int n);
17. void **printVector** (const double\* vector, const int n);
18. int **main** () {
19. int n = 11000;
20. double pi = 3.1415;
21. double\*\* matrix = createMatrix(n);
22. double\* uVector = createSinVector(n, pi);
23. double\* bVector = createVector(0, n);
24. mulMatrixVector(matrix, uVector, bVector, n);
25. double\* xVector = createVector(0, n);
26. double\* yVector = createVector(0, n);
27. double\* AyVector = createVector(0, n);
28. printf("expected answer - //%f//\n", scalarMul(bVector, bVector, n));
29. double e = 1;
30. double eps = 0.00001;
31. int cnt = 0;
32. time\_t start, end;
33. start = clock();
34. **while** (e > eps) {
35. cnt++;
36. e = updateX(matrix, bVector, xVector, yVector, AyVector, n);
37. }
38. end = clock();
39. double time = ((double) (end - start)) / CLOCKS\_PER\_SEC;
40. printf("time: %f\n", time);
41. printf("got answer - %lf\n", scalarMul(xVector, xVector, n));
42. deleteMatrix(matrix, n);
43. deleteVector(bVector);
44. deleteVector(uVector);
45. deleteVector(xVector);
46. deleteVector(AyVector);
47. deleteVector(yVector);
48. }
49. double\*\* **createMatrix** (const int n) {
50. double\*\* matrix = (double\*\*)malloc(n \* **sizeof**(double\*));
51. **for** (int i = 0; i < n; i++) {
52. matrix[i] = (double\*)malloc(n \* **sizeof**(double));
53. }
54. **for** (int i = 0; i < n; i++) {
55. **for** (int j = 0; j < n; j++) {
56. **if** (i == j) {
57. matrix[i][j] = 2.0;
58. }
59. **else** {
60. matrix[i][j] = 1.0;
61. }
62. }
63. }
64. **return** matrix;
65. }
66. void **deleteMatrix** (double\*\* matrix, const int n) {
67. **for** (int i = 0; i < n; i++) {
68. free(matrix[i]);
69. }
70. free(matrix);
71. }
72. double\* **createSinVector** (const int n, const double t) {
73. double\* vector = (double\*)malloc(n \* **sizeof**(double));
74. **for** (int i = 0; i < n; i++) {
75. vector[i] = sin((2 \* t \* i) / n);
76. }
77. **return** vector;
78. }
79. double\* **createZeroVector** (const int n) {
80. double\* vector = (double\*)calloc(n, **sizeof**(double));
81. **return** vector;
82. }
83. double\* **createVector** (const double t, const int n) {
84. double\* vector = (double\*)malloc(n \* **sizeof**(double));
85. **for** (int i = 0; i < n; i++) {
86. vector[i] = t;
87. }
88. **return** vector;
89. }
90. void **deleteVector** (double\* vector) {
91. free(vector);
92. }
93. void **mulMatrixVector**(double \*\*matrix, double \*vector, double \*res, const int n) {
94. setVectorZero(res, n);
95. #**pragma** omp parallel for
96. **for** (int i = 0; i < n; i++) {
97. **for** (int j = 0; j < n; j++) {
98. res[i] += matrix[i][j] \* vector[j];
99. }
100. }
101. }
102. void **sub** (double\* main\_vector, const double\* vector, const int n) {
103. #**pragma** omp parallel for
104. **for** (int i = 0; i < n; i++) {
105. main\_vector[i] -= vector[i];
106. }
107. }
108. void **mulByScalar** (double\* vector, double scalar, int n) {
109. #**pragma** omp parallel for
110. **for** (int i = 0; i < n; i++) {
111. vector[i] \*= scalar;
112. }
113. }
114. void **setVectorZero**(double\* vector, int n) {
115. **for** (int i = 0; i < n; i++) {
116. vector[i] = 0;
117. }
118. }
119. double **updateX** (double\*\* matrix, double\* bVector, double\* xVector, double\* yVector, double\* AyVector, const int n) {
120. mulMatrixVector(matrix, xVector, yVector, n);
121. sub(yVector, bVector, n);
122. double yNorm = sqrt(scalarMul(yVector, yVector, n));
123. double bNorm = sqrt(scalarMul(bVector, bVector, n));
124. mulMatrixVector(matrix, yVector, AyVector, n);
125. double tau = scalarMul(yVector, AyVector, n) / scalarMul(AyVector, AyVector, n);
126. mulByScalar(yVector, tau, n);
127. sub(xVector, yVector, n);
128. **return** yNorm / bNorm;
129. }
130. double **scalarMul** (const double\* vector1, const double\* vector2, const int n) {
131. double mult = 0;
132. #**pragma** omp parallel for reduction(+:mult)
133. **for** (int i = 0; i < n; i++) {
134. mult += (vector1[i] \* vector2[i]);
135. }
136. **return** mult;
137. }
138. void **printVector** (const double\* vector, const int n) {
139. printf("vector: ");
140. **for** (int i = 0; i < n; i++) {
141. printf("%f ", vector[i]);
142. }
143. printf("\n");
144. }

Приложение 2. 2ой вариант

1. #**include** <stdio.h>
2. #**include** <stdlib.h>
3. #**include** <math.h>
4. #**include** <time.h>
5. #**include** <omp.h>
6. double\*\* **createMatrix** (const int n);
7. void **deleteMatrix** (double\*\* matrix, const int m);
8. double\* **createSinVector** (const int n, const double t);
9. double\* **createVector** (const double val, const int maxM);
10. void **deleteVector** (double\* vector);
11. void **mulMatrixVector**(double \*\*matrix, double \*vector, double \*res, const int n);
12. void **sub** (double\* mainVector, const double\* vector, const int n);
13. void **mulByScalar** (double\* vector, double scalar, int n);
14. double **updateX** (double\*\* matrix, double\* bVector, double\* xVector, double\* yVector, double\* AyVector, const int n);
15. void **setVectorZero**(double\* vector, int n);
16. double **scalarMul** (const double\* vector1, const double\* vector2, const int n);
17. void **printVector** (const double\* vector, const int n);
18. void **solve**(double\*\* matrix, double\* bVector, double\* xVector, double\* yVector, double\* AyVector, const int n);
19. int **main** () {
20. int n = 11000;
21. double pi = 3.1415;
22. double\*\* matrix = createMatrix(n);
23. double\* uVector = createSinVector(n, pi);
24. double\* bVector = createVector(0, n);
25. mulMatrixVector(matrix, uVector, bVector, n);
26. double\* xVector = createVector(0, n);
27. double\* yVector = createVector(0, n);
28. double\* AyVector = createVector(0, n);
29. printf("expected answer - //%f//\n", scalarMul(bVector, bVector, n));
30. time\_t start, end;
31. start = clock();
32. solve(matrix, bVector, xVector, yVector, AyVector, n);
33. end = clock();
34. double time = ((double) (end - start)) / CLOCKS\_PER\_SEC;
35. printf("time: %f\n", time);
36. printf("got answer - %lf\n", scalarMul(xVector, xVector, n));
37. deleteMatrix(matrix, n);
38. deleteVector(bVector);
39. deleteVector(uVector);
40. deleteVector(xVector);
41. deleteVector(AyVector);
42. deleteVector(yVector);
43. }
44. void **solve**(double\*\* matrix, double\* bVector, double\* xVector, double\* yVector, double\* AyVector, const int n) {
45. double e = 1;
46. double eps = 0.00001;
47. int cnt = 0;
48. double yNormSquare = 0;
49. double bNormSquare = 0;
50. double tau = 0;
51. double tauPart1 = 0;
52. double tauPart2 = 0;
53. #**pragma** omp parallel
54. **while** (e > eps) {
55. // cnt++;
56. #**pragma** omp for
57. **for** (int i = 0; i < n; i++) {
58. yVector[i] = 0;
59. }
60. #**pragma** omp for
61. **for** (int i = 0; i < n; i++) {
62. **for** (int j = 0; j < n; j++) {
63. yVector[i] += matrix[i][j] \* xVector[j];
64. }
65. }
66. #**pragma** omp for
67. **for** (int i = 0; i < n; i++) {
68. yVector[i] -= bVector[i];
69. }
70. #**pragma** omp single
71. yNormSquare = 0;
72. #**pragma** omp single
73. bNormSquare = 0;
74. #**pragma** omp for reduction(+:yNormSquare)
75. **for** (int i = 0; i < n; i++) {
76. yNormSquare += (yVector[i] \* yVector[i]);
77. }
78. #**pragma** omp for reduction(+:bNormSquare)
79. **for** (int i = 0; i < n; i++) {
80. bNormSquare += (bVector[i] \* bVector[i]);
81. }
82. #**pragma** omp for
83. **for** (int i = 0; i < n; i++) {
84. AyVector[i] = 0;
85. }
86. #**pragma** omp for
87. **for** (int i = 0; i < n; i++) {
88. **for** (int j = 0; j < n; j++) {
89. AyVector[i] += matrix[i][j] \* yVector[j];
90. }
91. }
92. #**pragma** omp single
93. tauPart1 = 0;
94. #**pragma** omp single
95. tauPart2 = 0;
96. // double tau = scalarMul(yVector, AyVector, n) / scalarMul(AyVector, AyVector, n);
97. #**pragma** omp for reduction(+:tauPart1)
98. **for** (int i = 0; i < n; i++) {
99. tauPart1 += (yVector[i] \* AyVector[i]);
100. }
101. #**pragma** omp for reduction(+:tauPart2)
102. **for** (int i = 0; i < n; i++) {
103. tauPart2 += (AyVector[i] \* AyVector[i]);
104. }
105. #**pragma** omp single
106. tau = tauPart1 / tauPart2;
107. #**pragma** omp for
108. **for** (int i = 0; i < n; i++) {
109. yVector[i] \*= tau;
110. }
111. #**pragma** omp for
112. **for** (int i = 0; i < n; i++) {
113. xVector[i] -= yVector[i];
114. }
115. #**pragma** omp single
116. e = sqrt(yNormSquare) / sqrt(bNormSquare);
117. }
118. }
119. double\*\* **createMatrix** (const int n) {
120. double\*\* matrix = (double\*\*)malloc(n \* **sizeof**(double\*));
121. **for** (int i = 0; i < n; i++) {
122. matrix[i] = (double\*)malloc(n \* **sizeof**(double));
123. }
124. **for** (int i = 0; i < n; i++) {
125. **for** (int j = 0; j < n; j++) {
126. **if** (i == j) {
127. matrix[i][j] = 2.0;
128. }
129. **else** {
130. matrix[i][j] = 1.0;
131. }
132. }
133. }
134. **return** matrix;
135. }
136. void **deleteMatrix** (double\*\* matrix, const int n) {
137. **for** (int i = 0; i < n; i++) {
138. free(matrix[i]);
139. }
140. free(matrix);
141. }
142. double\* **createSinVector** (const int n, const double t) {
143. double\* vector = (double\*)malloc(n \* **sizeof**(double));
144. **for** (int i = 0; i < n; i++) {
145. vector[i] = sin((2 \* t \* i) / n);
146. }
147. **return** vector;
148. }
149. double\* **createVector** (const double t, const int n) {
150. double\* vector = (double\*)malloc(n \* **sizeof**(double));
151. **for** (int i = 0; i < n; i++) {
152. vector[i] = t;
153. }
154. **return** vector;
155. }
156. void **deleteVector** (double\* vector) {
157. free(vector);
158. }
159. void **mulMatrixVector**(double \*\*matrix, double \*vector, double \*res, const int n) {
160. setVectorZero(res, n);
161. #**pragma** omp parallel for
162. **for** (int i = 0; i < n; i++) {
163. **for** (int j = 0; j < n; j++) {
164. res[i] += matrix[i][j] \* vector[j];
165. }
166. }
167. }
168. void **sub** (double\* main\_vector, const double\* vector, const int n) {
169. #**pragma** omp parallel for
170. **for** (int i = 0; i < n; i++) {
171. main\_vector[i] -= vector[i];
172. }
173. }
174. void **mulByScalar** (double\* vector, double scalar, int n) {
175. #**pragma** omp parallel for
176. **for** (int i = 0; i < n; i++) {
177. vector[i] \*= scalar;
178. }
179. }
180. void **setVectorZero**(double\* vector, int n) {
181. **for** (int i = 0; i < n; i++) {
182. vector[i] = 0;
183. }
184. }
185. double **updateX** (double\*\* matrix, double\* bVector, double\* xVector, double\* yVector, double\* AyVector, const int n) {
186. mulMatrixVector(matrix, xVector, yVector, n);
187. sub(yVector, bVector, n);
188. double yNorm = sqrt(scalarMul(yVector, yVector, n));
189. double bNorm = sqrt(scalarMul(bVector, bVector, n));
190. mulMatrixVector(matrix, yVector, AyVector, n);
191. double tau = scalarMul(yVector, AyVector, n) / scalarMul(AyVector, AyVector, n);
192. mulByScalar(yVector, tau, n);
193. sub(xVector, yVector, n);
194. **return** yNorm / bNorm;
195. }
196. double **scalarMul** (const double\* vector1, const double\* vector2, const int n) {
197. double mult = 0;
198. #**pragma** omp parallel for reduction(+:mult)
199. **for** (int i = 0; i < n; i++) {
200. mult += (vector1[i] \* vector2[i]);
201. }
202. **return** mult;
203. }