МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Задание №1:
«Разработка параллельной версии программы RedBlack_2D с использованием технологии OpenMP»

325 группа ВМК МГУ
Бежанян Регина

2020 год

Содержание

Постановка задачи

Разработать параллельную версию программы для задачи RedBlack_2D с использованием технологии OpenMP, а затем исследовать масштабируемость полученной программы, построить графики зависимости времени её выполнения от числа используемых нитей и объёма входных данных. Сделать выводы с учетом полученных зависимостей.

Описание алгоритма

Алгоритм обрабатывает матрицу размера N*N, где N – объем данных, который задается до начала работы программы. Сначала происходит инициализация матрицы изначально заданным способом в функции init(). Дальше происходит основная часть работы алгоритма – релаксация матрицы relax(). Это итерационный метод решения систем линейных алгебраических уравнений. Последним шагом *verify()* вычисляется ответ.

Реализация параллельной версии программы с использованием технологии OpenMP

Для использования механизмов OpenMP необходимо подключить библиотеку <omp.h>.

Потенциал OpenMP реализуется полностью при использовании для поточной обработки самых трудоемких циклов. Поэтому распараллеливание необходимо для функции relax(). Но также оно было проведено в функциях init() и verify(). Однако две последние функции практически не влияют на общее время программы (были сделаны дополнительные подсчеты работы программы), так что это не необходимость.

Для замера времени использовался вызов функции omp_get_wtime.

Объем входных данных изменялся в самой программе, а количество используемых нитей нужно было ввести через аргументы командной строки.

Компиляция программы проводилась с помощью команды:

```
g++ -std=c++11 -Wall -fopenmp -o prog redblack_2d_openmp.cpp
```

Дальше в очередь подавался файл с прописаными командами:

```
Bsub < testFile.txt
```

Со следующим содержанием:

```
#BSUB -n 20 -q normal
#BSUB -W 15
#BSUB -o prog.%J.out
#BSUB -e prog.%J.err
echo 1 thread:
./prog 1
echo 2 threads:
./prog 2
echo 4 threads:
./prog 4
echo 8 threads:
./prog 8
```

echo 16 threads:

./prog 16

echo 32 threads:

./prog 32

echo 64 threads:

./prog 64

echo 128 threads:

./prog 128

echo 160 threads:

./prog 160

Информация о времени работы программы при разном количестве нитей извлекалась из файлов prog.%J.out.

Код программы:

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>
#define  Max(a,b) ((a)>(b)?(a):(b))

#define  N   2048
double   maxeps = 0.1e-7;
     int itmax = 100;
     int i,j,k;
     double w = 0.5;
     double eps;
     int num_thr = 0;

     double A [N][N];

     void relax();
     void init();
     void verify();

```

```
22.
23.     int main(int argc, char **argv){
24.
25.         if (argc < 2) {
26.             printf("Too few arguments\n");
27.             return -1;
28.         }
29.
30.         num_thr = atoi(argv[1]);
31.         int it;
32.         init();
33.         double start_time = omp_get_wtime();
34.         for(it = 1; it <= itmax; it++){
35.             eps = 0.;
36.             relax();
37.             if (eps < maxeps)
38.                 break;
39.         }
40.         double end_time = omp_get_wtime();
41.         verify();
42.         printf("number of threads: %d\n time: %f\n", num_thr,
    end_time - start_time);
43.         return 0;
44.     }
45.
46.     void init(){ //инициализация матрицы
47.         #pragma omp parallel shared(A) num_threads(num_thr)
48.         {
49.             #pragma omp for private(i,j)
50.             for(i = 0; i <= N-1; i++){
51.                 for(j = 0; j <= N-1; j++){
52.                     if(i == 0 || i == N-1 || j == 0 || j == N-
    1) A[i][j]= 0.;
53.                     else A[i][j]= ( 1. + i + j) ;
54.                 }
55.             }
56.         }
57.     }
58.
59.
```

```
60.     void relax(){ //релаксация матрицы
61.         #pragma omp parallel shared(A) num_threads(num_thr)
   reduction(max: eps)
62.         {
63.             #pragma omp for private(i,j)
64.             for(int i = 1; i <= N-2; i++){
65.                 for(int j = 1 + i % 2; j <= N-2; j+=2){
66.                     double b;
67.                     b = w * ((A[i-1][j] + A[i+1][j] + A[i][j-1]
   + A[i][j+1]) / 4. - A[i][j]);
68.                     eps = Max(fabs(b),eps);
69.                     A[i][j] = A[i][j] + b;
70.                 }
71.             }
72.             #pragma omp for private(i,j)
73.             for(int i=1; i <= N-2; i++){
74.                 for(int j = 1 + (i + 1) % 2; j<=N-2; j+=2){
75.                     double b;
76.                     b = w * ((A[i-1][j] + A[i+1][j] + A[i][j-1]
   + A[i][j+1]) / 4. - A[i][j]);
77.                     A[i][j] = A[i][j] + b;
78.                 }
79.             }
80.             #pragma omp critical
81.             {
82.                 eps = Max(eps, eps);
83.             }
84.
85.         }
86.     }
87.
88.
89.     void verify(){ //вычисление ответа
90.         double s;
91.         s = 0.;
92.         #pragma omp parallel shared(A) num_threads(num_thr)
   reduction(+: s)
93.         {
94.             #pragma omp for private(i,j)
95.             for(i = 0; i <= N-1; i++){
```

```
96.                 for(j = 0; j <= N-1; j++){
97.                     s = s + A[i][j] * (i+1) * (j+1) / (N*N);
98.                 }
99.             }
100.        }
101.        printf("S = %f\n",s);
102.    }
```

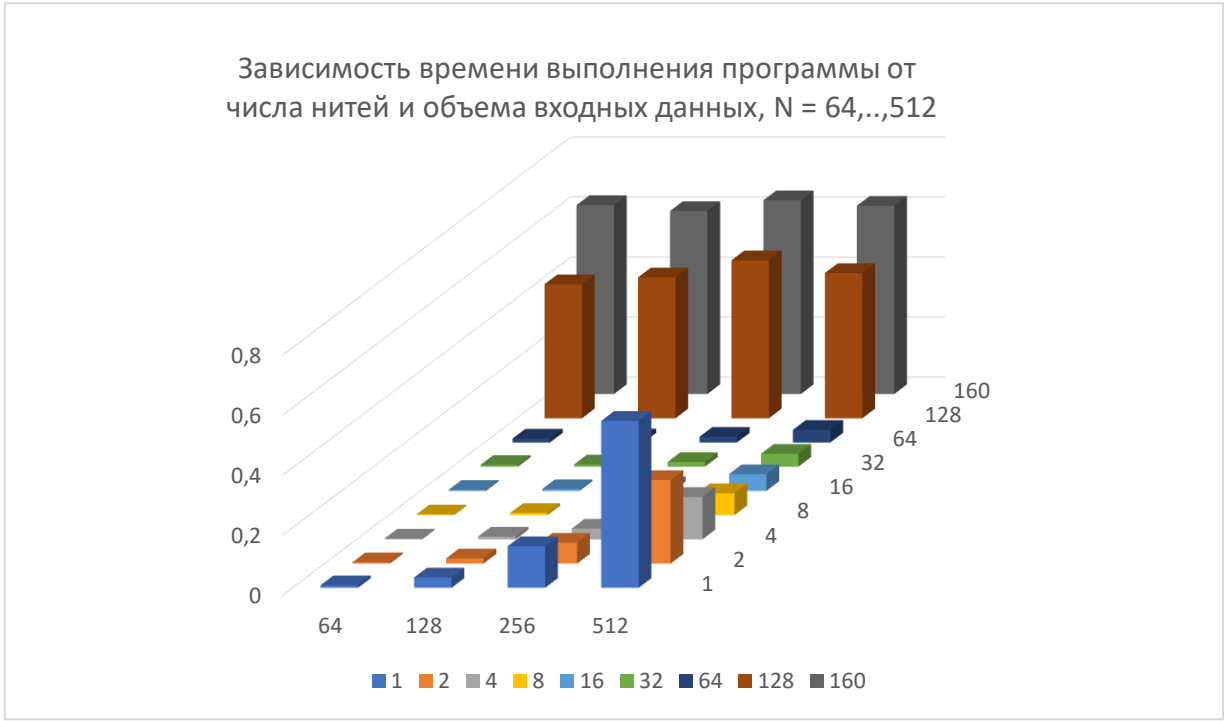
Результаты

Для подсчета времени выполнения производилось три запуска на одинаковых данных, после чего было усреднение результат для каждого числа нитей. Это производится для избавления от случайных выбросов.

Время выполнения

Число нитей, num_thr	Размер массива, N							
	64	128	256	512	1024	2048	4096	8192
1	0,008592	0,034465	0,138785	0,556226	2,229134	8,927473	36,06517	143,2469
2	0,004529	0,017485	0,069592	0,278599	1,124549	4,467585	18,35495	71,78977
4	0,002871	0,009452	0,035838	0,141492	0,563713	2,254636	10,92912	36,06098
8	0,002661	0,007245	0,019442	0,073197	0,288264	1,158193	4,783301	18,46019
16	0,003552	0,006065	0,015766	0,056043	0,217454	0,86698	3,361603	12,96644
32	0,005995	0,007515	0,015725	0,043065	0,170343	0,645643	2,551961	9,480552
64	0,013184	0,013957	0,018928	0,042125	0,142905	0,512272	2,176741	7,810574
128	0,446701	0,469528	0,526464	0,483738	0,367935	0,428714	1,356546	5,98559
160	0,629728	0,609936	0,645848	0,62752	0,623266	0,753995	1,678852	5,185043

Для наглядности построим несколько графиков:


Зависимость времени выполнения программы от числа нитей и объема входных данных, N = 64,..,512
0,8
0,6
0,4
0,2
0
64
128
256
512
1
2
4
8
16
32
64
128
160
1 2 4 8 16 32 64 128 160

Выводы

На основе полученных результатов и построенных графиков можно сделать вывод о том, что при большом N = 2048,..,8192 при увеличении числа нитей производительность увеличивается. Для N = 2048, 4096 оптимальное число нитей – 128, для N = 8192 – можно брать больше число.

Для относительно малых N = 64,..,512 оптимальное число нитей – 4 – 16. Также этом можно заметить, что при num_thr >= 128 время работы программы резко увеличивается. Это связано с большими накладными расходами на создание новых нитей.