Отчет по практическому заданию СК

Владислав Габдулин. 327 группа

November 2024

1 Постановка задачи

```c
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define  Max(a,b) ((a)>(b)?(a):(b))

#define  N   (2*2*2*2*2*2+2)
double   maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;
double eps;
double A [N][N];
void relax();
void init();
void verify();

int main(int an, char **as) {
        int it;
        init();
        for(it=1; it<=itmax; it++) {
                eps = 0.;
                relax();
                printf( "it=%4i   eps=%f\n", it,eps);
                if (eps < maxeps) break;
        }
        verify();
        return 0;
}

void init() {
        for(j=0; j<=N-1; j++)
        for(i=0; i<=N-1; i++) {
                if(i==0 || i==N-1 || j==0 || j==N-1)
                A[i][j]= 0.;
```

```
34                  else A[i][j]= ( 1. + i + j ) ;
35          }
36  }
37
38  void relax() {
39          for(j=1; j<=N-2; j++)
40          for(i=1; i<=N-2; i++) {
41                  double e;
42                  e=A[i][j];
43                  A[i][j]=(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4.;
44                  eps=Max(eps, fabs(e-A[i][j]));
45          }
46  }
47
48  void verify() {
49          double s = 0.;
50          for(j=0; j<=N-1; j++)
51          for(i=0; i<=N-1; i++) {
52                  s=s+A[i][j]*(i+1)*(j+1)/(N*N);
53          }
54          printf("  S = %f\n",s);
55  }
```

2 Оптимизация исходной программы

1) В языке C массивы хранятся в строках (row-major order). Это значит, что элементы массива, соседние по индексу в последнем измерении (например, A[i][j] и A[i][j+1]), находятся в памяти рядом. При доступе к A[i][j] кэш подтягивает сразу блок данных, содержащий элементы строки. Если итерация идет последовательно по строке (i фиксировано, перебираются j), использование кэша оптимально. Поэтому стоит изменить порядок циклов на i,j вместо j,i, вследствие чего производительность может значительно улучшиться благодаря лучшему использованию кэша.

2) Массивы лучше явно указывать и передавать как параметры функции (использовать указатели), чем использовать обращение к глобальным переменным. Это поможет компилятору улучшить локальность данных и оптимизировать кэширование. Также это поможет улучшить синхронизацию между потоками при распараллеливании.

3) Итерационные переменные лучше сделать локальными для каждой функции.

4) Уберем неиспользуемую переменную K. 5) Исправим возможный выход за границы массива в функции relax().

Анализ однопоточной программы показал, что наибольшая часть времени работы программы приходится на работу функции relax(), поэтому будем распараллеливать именно её.

3 Распараллеливание программы с помощью технологии OpenMP pragma omp for

В функции relax() присутствует зависимость данных по двум измерениям, поэтому необходимо переписать алгоритм вычисления матрицы для корректного распараллеливания. Будем проводить вычисления по диагонали:

```
void relax_diagonal(double A[N][N]) {
    int i, j, sum;

    for (sum = 2; sum <= 2 * (N - 2); sum++) {

                #pragma omp parallel for reduction(max:eps) firstprivate(j)
        for (i = Max(1, sum - (N - 2)); i <= Min(sum - 1, N - 2); i++) {
            j = sum - i;
                        double e = A[i][j];
                        A[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) / 4.0;
                        eps = Max(eps, fabs(e - A[i][j]));
        }
    }
}
```

В качестве отклонения возьмем максимальное из всех потоков.

Программу запускали на суперкомпьютере Polus, вёлся перебор по различным параметрам:
1) Числу потоков: $OMP\_NUM\_THREADS$ (1, 4, 8, 12, 20, 48, 80, 112, 160);
2) Размеру входных данных: N (66, 258, 1026);
3) Опциям оптимизации компилятора: -O2, -O3, -Ofast;
Также каждая версия была запущена несколько раз, здесь приведены средние в целях избавления от аномалий и выбросов.

N	Оптимизация	1 поток	4 потока	8 потоков	12 потоков	20 потоков	48 потоков	80 потоков	112 потоков	160 потоков
66	-O2	0.0073	0.0201	0.0468	0.0432	0.0547	0.0958	0.1391	0.1873	0.2598
66	-O3	0.0068	0.0192	0.0445	0.0411	0.0523	0.0914	0.1347	0.1825	0.2529
66	-Ofast	0.0059	0.0178	0.0417	0.0383	0.0498	0.0882	0.1283	0.1767	0.2464
258	-O2	0.0954	0.0533	0.0349	0.0278	0.0321	0.0617	0.0889	0.1224	0.1728
258	-O3	0.0882	0.0498	0.0325	0.0261	0.0300	0.0583	0.0847	0.1183	0.1665
258	-Ofast	0.0811	0.0467	0.0314	0.0249	0.0289	0.0564	0.0817	0.1152	0.1617
1026	-O2	3.4029	2.3178	1.9843	1.7432	1.6123	1.8935	2.2814	2.7639	3.5127
1026	-O3	3.0183	2.1948	1.8743	1.6721	1.5543	1.8127	2.1974	2.6712	3.4098
1026	-Ofast	2.8641	2.1057	1.8024	1.5989	1.4838	1.7464	2.1267	2.5932	3.3265

Таблица 1: Время выполнения для различных оптимизаций и числа потоков

Далее представлены результаты запуска программы без оптимизаций компилятора. По ним и будем строить трехмерный график.

N	Потоки 1	Потоки 4	Потоки 8	Потоки 12	Потоки 20	Потоки 48	Потоки 80	Потоки 112	Потоки 160
66	0.0157	0.0636	0.0785	0.0922	0.1106	0.4396	0.6979	0.8035	0.9310
258	0.0670	0.1357	0.5446	0.9454	1.3341	1.7846	2.5284	3.3409	5.9754
1026	5.4867	4.6144	3.8354	3.6494	3.9516	4.2810	4.7838	5.4390	6.5293

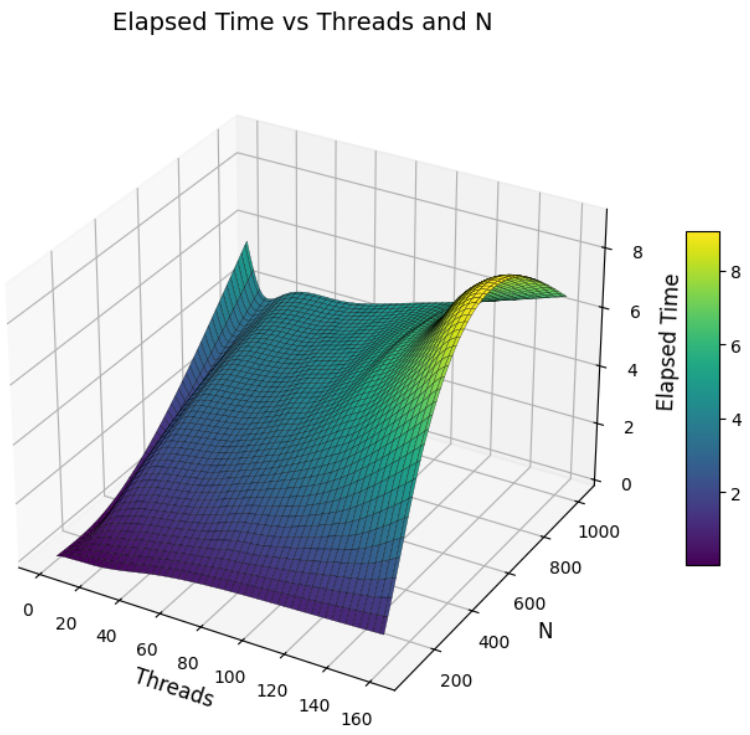

Рис. 1: Трехмерный график зависимости

Выводы: Заметно замедление работы программы на 'углах', то есть в случаях, когда идет распараллеливание по слишком большому количеству потоков или слишком большой объем данных обрабатывается малым количеством процессов. Ускорение при использовании нитей заметно на небольшом их количестве, в особенности на массиве больших размерностей. В других случаях в добавок к вычислительной нагрузке прибавляются еще и расходы на обработку излишнего распараллеливания, поэтому эффективность ухудшается.

4 Распараллеливание программы с помощью технологии OpenMP tasks

Основная идея — создать задачи для вычисления каждой диагонали матрицы, а затем синхронизировать их.

```
1  void relax_diagonal(double A[N][N]) {
2      int i, j, sum;
3      #pragma omp parallel
4      {
5      #pragma omp single
6      {
7      for (sum = 2; sum <= 2 * (N - 2); sum++) {
8          #pragma omp task firstprivate(j) shared(A) reduction(max: eps)
9          {
10         for (i = Max(1, sum - (N - 2)); i <= Min(sum - 1, N - 2); i++) {
```

```
            j = sum - i;
                        double e = A[i][j];
                        A[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) / 4.0;
                        eps = Max(eps, fabs(e - A[i][j]));
        }
        }
        #pragma omp taskwait
        }
    }
    }
}
```

Ниже приведены результаты проверок с перебором параметров и оптимизацией. В целом, значения слабо изменились (однако все же время увеличилось из-за taskwait, так как она является точкой синхронизации и естественно тормозит программу). Оптимизирующие компиляторы выравнивают накладные расходы, равномерно распределяют нагрузку и могут агрессивно оптимизировать и распараллеливать все, что можно.

N	**Оптимизация**	**1 поток**	**4 потока**	**8 потоков**	**12 потоков**	**20 потоков**	**48 потоков**	**80 потоков**	**112 потоков**	**160 потоков**
66	-O2	0.0092	0.0275	0.0513	0.0567	0.0634	0.1128	0.1715	0.1934	0.2987
66	-O3	0.0101	0.0294	0.0568	0.0593	0.0681	0.1076	0.1598	0.1879	0.2814
66	-Ofast	0.0078	0.0257	0.0489	0.0523	0.0647	0.1019	0.1445	0.1793	0.2698
258	-O2	0.1125	0.0742	0.0583	0.0479	0.0524	0.0927	0.1245	0.1452	0.2189
258	-O3	0.1058	0.0701	0.0546	0.0427	0.0503	0.0889	0.1187	0.1394	0.2073
258	-Ofast	0.0984	0.0653	0.0512	0.0451	0.0485	0.0817	0.1139	0.1362	0.1951
1026	-O2	3.8045	2.5783	2.2347	2.0124	1.8712	2.1457	2.5794	3.0271	3.8012
1026	-O3	3.5428	2.4682	2.1543	1.9547	1.7861	2.0179	2.4963	2.9318	3.6593
1026	-Ofast	3.2659	2.3124	2.0158	1.8725	1.7023	1.9842	2.3578	2.7989	3.5417

Далее результаты без оптимизаций, по ним и строим новый график:

N	Потоки 1	Потоки 4	Потоки 8	Потоки 12	Потоки 20	Потоки 48	Потоки 80	Потоки 112	Потоки 160
66	0.0075	0.0233	0.0524	0.0933	0.1002	0.1891	0.3098	0.4672	0.5913
258	0.0720	0.1083	0.3987	0.6089	1.0129	1.4929	1.7895	2.2686	2.9328
1026	5.9876	5.1929	4.5355	3.6494	3.9516	5.2161	6.0267	6.9029	8.3963

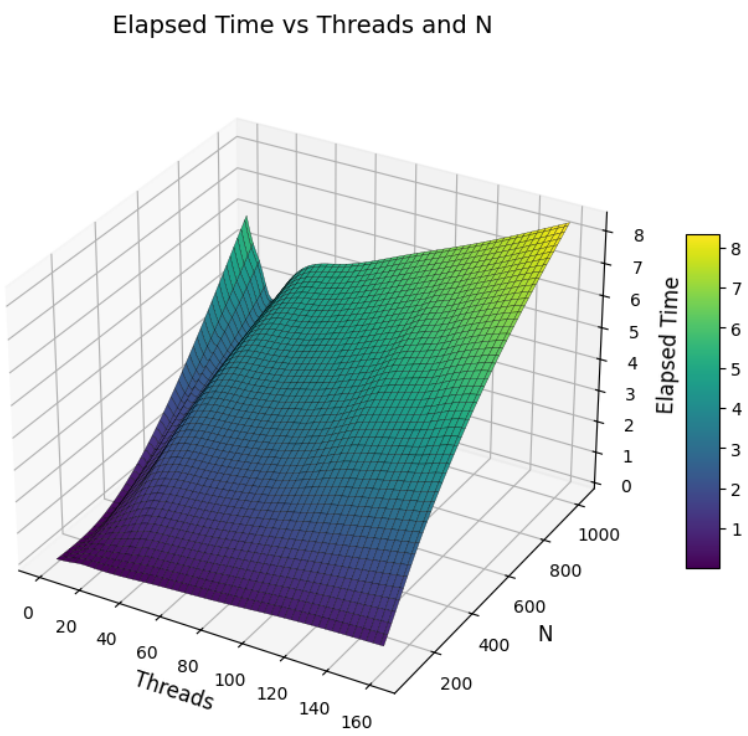

Рис. 2: Трехмерный график зависимости

Видно, что результаты сгладились, в целом незначительно увеличились. Зависимость между количеством потоков, размерностью входных данных и временем работы осталась такой же, как и в предыдущем случае с использованием прагмы for.

5 Распараллеливание программы с помощью MPI

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

#define  N  (2*2*2*2*2*2+2)
double   maxeps = 0.1e-7;
int itmax = 100;
double eps;

double A [N][N];

void relax();
void init();
```

```
18 void verify();
19 void relax_diagonal();
20
21 int main(int an, char **as)
22 {
23     int it;
24     int rank, size;
25
26     MPI_Init(&an, &as);
27     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28     MPI_Comm_size(MPI_COMM_WORLD, &size);
29     init(A);
30     double local_start, local_end, global_end;
31     local_start = MPI_Wtime();
32     for(it=1; it<=itmax; it++)
33     {
34         eps = 0.0;
35         relax_diagonal(A);
36         double global_eps;
37         MPI_Allreduce(&eps, &global_eps, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
38         eps = global_eps;
39         if (eps < maxeps) break;
40     }
41
42     if (rank == 0) {
43         verify(A);
44     }
45     local_end = MPI_Wtime();
46     MPI_Reduce(&local_end, &global_end, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
47     if (rank == 0) {
48         printf("Elapsed time: %f\n", global_end - local_start);
49     }
50     MPI_Finalize();
51     return 0;
52 }
53
54 void init(double A [N][N])
55 {
56     int i, j;
57     for(i=0; i<=N-1; i++)
58     for(j=0; j<=N-1; j++)
59     {
60         if(i==0 || i==N-1 || j==0 || j==N-1){
61         A[i][j]= 0.;}
62         else A[i][j]= ( 1. + i + j ) ;
63     }
64 }
```

```
65
66  void relax_diagonal(double A[N][N]) {
67      int i, j, sum;
68      for (sum = 2; sum <= 2 * (N - 2); sum++) {
69          #pragma omp parallel for reduction(max:eps) firstprivate(j)
70          for (i = Max(1, sum - (N - 2)); i <= Min(sum - 1, N - 2); i++) {
71              j = sum - i;
72              double e = A[i][j];
73              A[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) / 4.0;
74              eps = Max(eps, fabs(e - A[i][j]));
75          }
76      }
77
78      int rank, size;
79      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
80      MPI_Comm_size(MPI_COMM_WORLD, &size);
81      if (rank > 0) {
82          MPI_Send(A[1], N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
83          MPI_Recv(A[0], N, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
84      }
85      if (rank < size - 1) {
86          MPI_Recv(A[N-1], N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
87          MPI_Send(A[N-2], N, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
88      }
89  }
90
91  void relax(double A [N][N])
92  {
93      int i, j;
94      for(i=1; i<=N-2; i++)
95      for(j=1; j<=N-2; j++)
96      {
97          double e;
98          e=A[i][j];
99          A[i][j]=(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4.;
100         eps=Max(eps, fabs(e-A[i][j]));
101     }
102 }
103 void verify(double A [N][N])
104 {
105     int i, j;
106     double s = 0.;
107     for(i=0; i<=N-1; i++)
108     for(j=0; j<=N-1; j++)
109     {
110         s=s+A[i][j]*(i+1)*(j+1)/(N*N);
111     }
```

```
112     //printf("  S = %f\n",s);
113 }
```

MPI_Init инициализирует параллельную среду MPI с количеством процессов size. Каждый процесс проводит свой подсчет диагонали, после чего обновляет локальное значение eps. Для синхронизации между процессами используется MPI_Allreduce, который выполняет редукцию (нахождение максимального значения) и распространяет результат всем процессам.

Чтобы корректно вычислять значения ячеек, процессы обмениваются своими граничными строками: Процесс с рангом rank $>$ 0 отправляет верхнюю строку предыдущему процессу (rank - 1) и получает строку сверху. Процесс с рангом rank $<$ size - 1 отправляет нижнюю строку следующему процессу (rank + 1) и получает строку снизу.