Студент: [redacted]

Отчет о выполнении задания по параллельному программированию

Вариант 15

Исходный код программы приведен ниже:

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

#define  N   (2*2*2*2*2*2+4)
double   maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;
double eps;
double A [N][N][N],  B [N][N][N];

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

int main(int an, char **as)
{
          int it;
          init();
          clock_t begin = clock();
          for(it=1; it<=itmax; it++)
          {
                    eps = 0.;
                    relax();
                    resid();
                    printf( "it=%4i   eps=%f\n", it,eps);
                    if (eps < maxeps) break;
          }
          clock_t end = clock();
          double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
          verify();
          printf("Time programs = "%f\n", time_spent);
          printf("%d\n", N);
          return 0;
}

void init()
{
          for(k=0; k<=N-1; k++)
          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 || k==0 || k==N-1) A[i][j][k]= 0.;
                    else A[i][j][k]= ( 4. + i + j + k) ;
          }
}

void relax()
{
          for(k=2; k<=N-3; k++)
          for(j=2; j<=N-3; j++)
          for(i=2; i<=N-3; i++)
          {
                    B[i][j][k]=(A[i-1][j][k]+A[i+1][j][k]+A[i][j-1][k]+A[i][j+1][k]+A[i][j][k-1]+A[i][j][k+1]+
                              A[i-2][j][k]+A[i+2][j][k]+A[i][j-2][k]+A[i][j+2][k]+A[i][j][k-2]+A[i][j][k+2])/12.;
          }
}

void resid()
{
          for(k=2; k<=N-3; k++)
          for(j=2; j<=N-3; j++)
          for(i=2; i<=N-3; i++)
          {
                    double e;
                    e = fabs(A[i][j][k] - B[i][j][k]);
                    A[i][j][k] = B[i][j][k];
                    eps = Max(eps,e);
          }
}

void verify()
{
          double s;
          s=0.;
          for(k=0; k<=N-1; k++)
          for(j=0; j<=N-1; j++)
          for(i=0; i<=N-1; i++)
          {
                    s=s+A[i][j][k]*(i+1)*(j+1)*(k+1)/(N*N*N);
          }
          printf("  S = %f\n",s);
}
```

1. Оптимизация исходной программы

1) Для ускорения программы без применения технологий распараллеливания в первую очередь стоит поменять порядок переменных в циклах (с k, j, i на i, j, k) – это позволит ускорить программу за счет оптимизации доступа к памяти.
2) При передаче указателей на массивы через функции данные могут быть локально закэшированы. Когда функция работает с массивом, его элементы могут оставаться в кэше процессора, что уменьшает задержку при доступе к памяти. В случае глобальных указателей на массивы такой эффект может быть утрачен. Поэтому массивы A, B будем передавать в функции явно.
3) Если посмотреть внимательно на функции relax() и resid(), то можно увидеть, что выполняются одинаковые проходы по циклам, т.е. можно объединить эти функции, обойдя лишние расходы.

```
void relax(double A[N][N][N], double B[N][N][N])
{
        for(i = 2; i <= N-3; i++)
        for(j = 2; j <= N-3; j++)
        for(k = 2; k <= N-3; k++)
        {
                        B[i][j][k] = (A[i-1][j][k] + A[i+1][j][k] + A[i][j-1][k] +A[i][j+1][k]+
 A[i][j][k-1] + A[i][j][k+1] + A[i-2][j][k] + A[i+2][j][k] + A[i][j-2][k] +
 A[i][j+2][k] + A[i][j][k-2] + A[i][j][k+2]) / 12.;
                        double e;
                        e = fabs(A[i][j][k] - B[i][j][k]);
                        eps = Max(eps,e);
        }
}
```

При этом в функции `main()` будем менять местами передачу массивов A и B при вызове функции `relax()`:

```
        if(it%2) relax(A, B);
        else relax(B, A);
```

4) Сделаем переменные i, j, k локальными для каждой функции, одни должны быть приватными для каждого цикла.

Сделаем замеры времени работы программы с различными размерами переменной N (в данной задаче она является показателем объёма входных данных) на машине K-10 с помощью компилятора icc.

Таблица 1. Времена (сек.) исходной и оптимизированной программ при различных N

Программа \ **N**	68	132	260	516
Исходная	0.400000	4.370000	54.890000	800.410000
Оптимизированная	0.112772	0.935050	7.472822	60.608663

Из таблицы видно, что ускорение существенно и с увеличением N только возрастает.

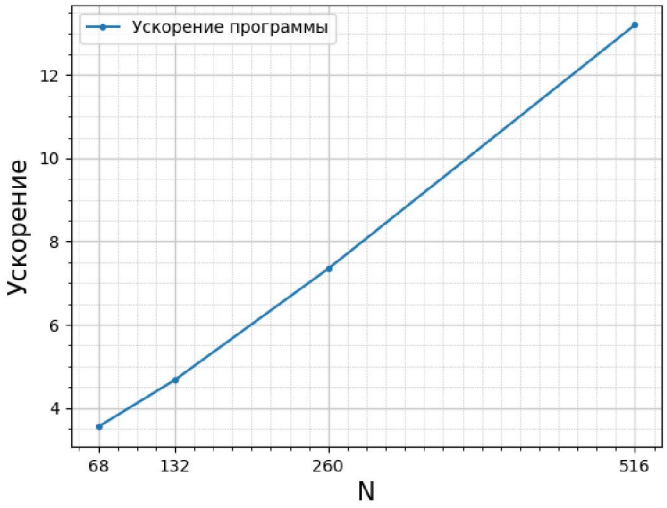


График1. Ускорения оптимизированной программы в зависимости от N.

2. Распараллеливание программы с помощью технологии OpenMP #pragma omp for

При анализе программы было выявлено, что 98% времени уходит на вызовы функции relax(), поэтому будем распараллеливать именно её часть кода.

В функции relax() находится 3 вложенных цикла без зависимости по данным. Будем применять возможности технологии openMP `#pragma omp parallel for.` Переменные i ,j, k у каждого потока должны свои, поэтому делаем их приватными, массивы A и B сделаем общими. В конце цикла идет сравнение глобальной переменной eps с вычисленной на

Получившееся прагма:

Сделав замер времени полученной программы при N=68 получим результат 0.024631 сек. на 8 нитях. Теперь применим `#pragma omp parallel for default(none) shared(A, B) private(i,j,k)` к вложенным циклам функции init() и получим замер времени 0.015556 секунд. Получили ускорение на 45% в данном случае. Это связанно архитектурой доступа памяти вычислительной машины.

Теперь сделаем замеры времени работы полученной программы на разных количествах нитей и различном объеме входных данных представим результаты в виде таблицы.

Таблица 2. Время работы (сек.) программы при различных числах нитей и N

Число нитей / N	1	2	4	8
68	0.112772	0.056656	0.028900	0.015556
132	0.935050	0.472909	0.224877	0.116314
260	7.472822	3.774928	1.902570	0.987524
516	60.608663	30.937738	18.786623	14.964459

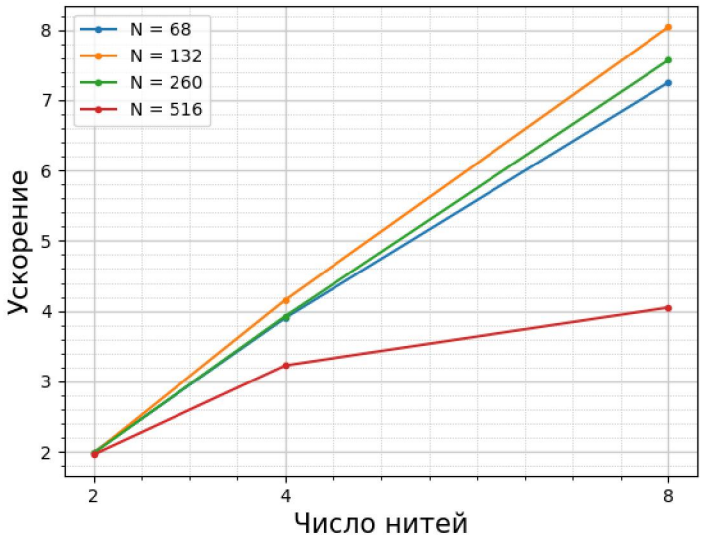


График 2. Сравнение ускорений программы при различных числах нитей.

На графике можем увидеть хорошую масштабируемость на N = 68, 132, 260. На N=68 на 4 и 8 нитях наблюдается сверхлинейное ускорение (причиной является работа с кэш-памятью). При N=516 на 8 нитях падает производительность (предположительно из-за неравномерной загрузки процессоров)

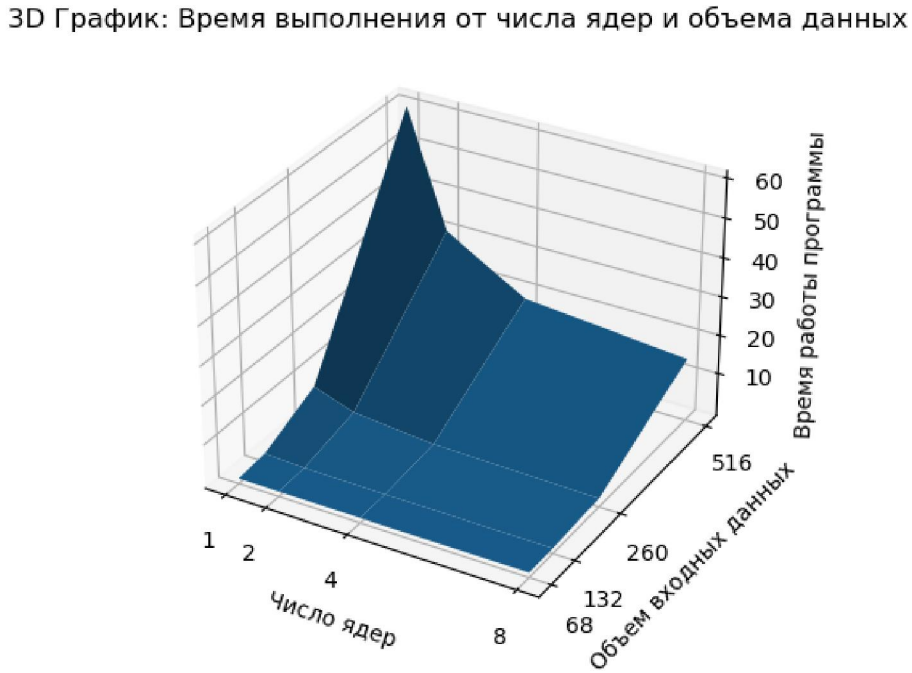


График 3. Время выполнения программы в зависимости от числа нитей и объема данных.

3. Распараллеливание программы с помощью технологии OpenMP #tasks

В функции main() создадим параллельную область `#pragma omp parallel shared(A,B, eps)` с выполнением одного потока

```
int main(int an, char **as)
{
    ...
#pragma omp parallel shared(A,B, maxres, eps)
{
   #pragma omp single
   {
   ...
   }
}
...
}
```

Далее в функции relax() применяем технологию task, приписывая вложенным циклам прагму

которая автоматически распределяет задачи между вложенными циклами. Здесь `#pragma omp taskloop` - директива OpenMP, которая указывает, что итерации цикла будут выполнены в виде задач, а параметр `grainsize(TS)` контролирует, сколько логических итераций цикла будет назначено каждой созданной задаче, `int TS = (N - 4) / omp_get_max_threads()`.

Однако, есть некоторые сложности:

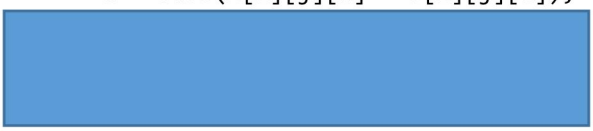
1) На К-10 старая версия компилятора, которая не позволяет использовать клаузу reduction.

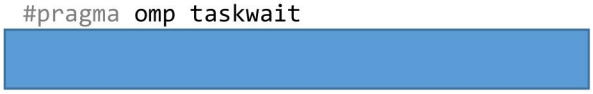
Вместо этого создадим динамически глобальный массив размера количества потоков

каждый элемент которого будет содержать максимальную eps для своего номера нити, после выполнения задач в данном массиве находится максимальный элемент, который присваивается eps. Перед каждой итерации в функции main() элементы обнуляются.

Функция relax получается следующей:

```
void relax(double A[N][N][N], double B[N][N][N], double* maxres)
{
    int i, j, k, ij, iji, UB;
    int TS = (N - 4) /nthrds;
```

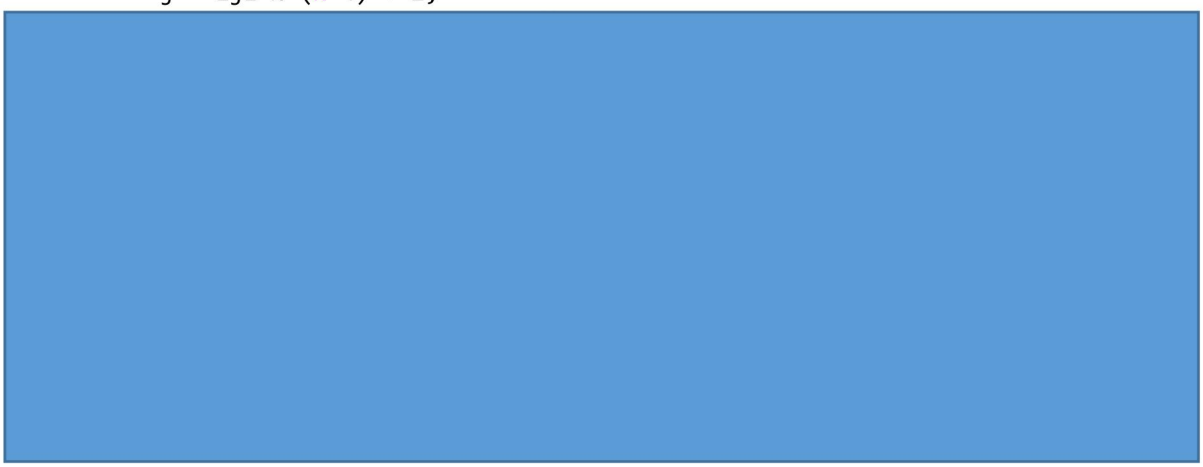
```
            e = fabs(A[i][j][k] - B[i][j][k]);
```

```
    }
}
```

```
    #pragma omp taskwait
```

```
}
```

#pragma omp taskwait заставляет выполнить все поставленные задачи.

2) Но при запуске данной реализации замер времени на 1 нити при N=68 показывает 0.266795 сек., что больше чем в 2 раза реализации #pragma omp for. Дело в том, что конструкция #pragma omp taskloop используется для указания, что итерации одного или нескольких связанных циклов будут выполняться параллельно с использованием явных задач, что в нашей реализации замедляет процесс. В связи с этим grainsize(TS) collapse(2) реализуем вручную.

Получим в результате следующую функцию relax():

```
void relax(double A[N][N][N], double B[N][N][N], double* maxres)
{
    int i, j, k, ij, iji, UB;
    int TS = (N - 4) /nthrds;
    int Size = (N-4) * (N-4);
    for(ij = 0; ij < Size; ij+=TS)
    {
       int UB = Size < (ij + TS)? Size:ij + TS;
#pragma omp task private(i,j,k,iji) firstprivate(ij, UB)
{
       for(iji = ij; iji < UB; iji++)
       {
           i = iji / (N-4) + 2;
           j = iji % (N-4) + 2;
```

```
    #pragma omp taskwait
```

```
}
```

Теперь сделаем замеры времени работы полученной программы на разных количествах нитей и различном объеме входных данных представим результаты в виде таблицы.

Таблица 3. Время работы (сек.) программы при различных числах нитей и N (tasks)

N \ Число нитей	1	2	4	8
68	0.153107	0.088643	0.064815	0.116631
132	1.053227	0.585896	0.338297	0.238416
260	8.190187	4.279491	2.704566	1.912407
516	68.482210	36.938582	18.622620	15.508463

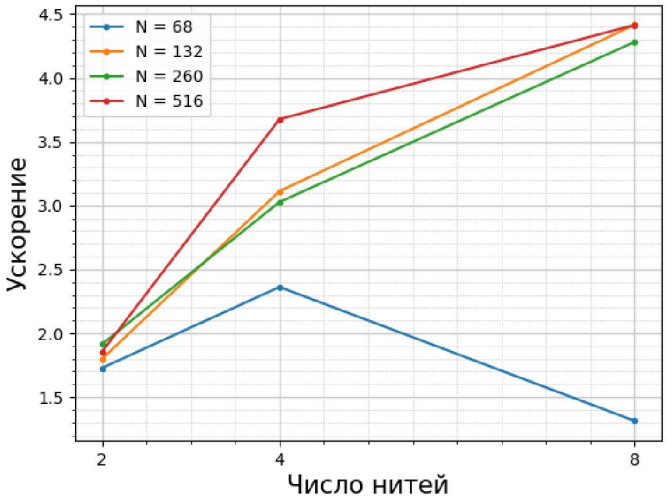


График 4. Сравнение ускорений программы (tasks) при различных числах нитей.

На графике можем увидеть, как при 8 нитях на задаче N=68 производительность падает ниже работы программы на 1 нити. Это связано со слишком маленькими размерами задач и частым переключением между ними. Однако, с увеличением задач данная ситуация меняется в лучшую сторону, что свидетельствует хорошая слабая масштабируемость.

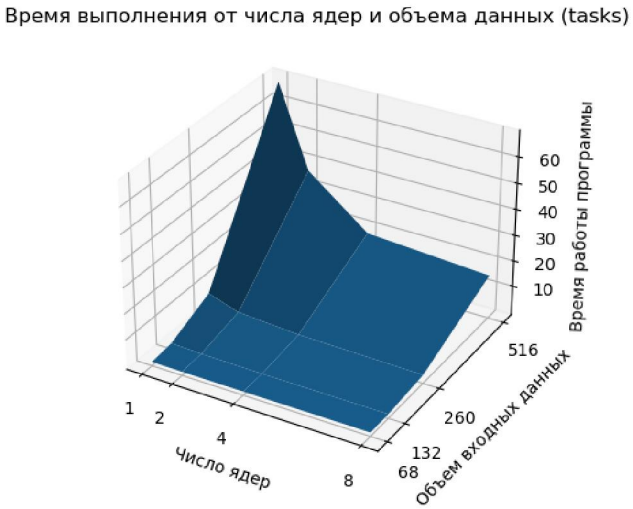


График 5. Время выполнения программы (tasks) в зависимости от числа нитей и объема данных.

Результаты, полученные после распараллеливания программы с использованием OpenMP директив for и task, оказались сопоставимыми. Этим можно убедиться, глядя на построенные графики 4 и 5.