СУПЕРКОМПЬЮТЕРНЫЙ КОНСОРЦИУМ УНИВЕРСИТЕТОВ РОССИИ

МОЛОДЕЖНАЯ ШКОЛА
«Суперкомпьютерные технологии в науке, промышленности и образовании»

Секция: «Параллельное программирование в системах с общей памятью»

Курс: «Технологии параллельного программирования: OpenMP»

Лабораторная работа: «Разработка параллельных программ с использованием технологии OpenMP»

Бахтин Владимир Александрович
к.ф.-м.н., зав. сектором Института прикладной математики им М.В.Келдыша РАН

ННГУ им. Н.И. Лобачевского, Нижний Новгород, 26-31 октября 2009 г.

Содержание

- Распараллеливание итерационных методов
  - Алгоритм Якоби
  - Алгоритм последовательной верхней релаксации
    - Распараллеливание циклов с зависимость по данным
- Балансировка нагрузки

Цель работы

Лабораторная работа направлена на достижение следующих целей:

- Получить навыки распараллеливания существующих последовательных программ на языке Си.
- Оценить производительность параллельной программы при использовании разного количества процессоров (ядер).
- Для определенного количества процессоров (ядер) получить максимальную производительность, изменяя способ распределения витков циклов по процессорам (используя конструкцию SCHEDULE).

Система линейных уравнений

Рассмотрим следующую систему линейных уравнений:

A x = b

где A – матрица коэффициентов, b – вектор свободных членов, x – вектор неизвестных.

Для решения этой системы можно использовать следующие методы.

Прямые методы. Хорошо известный метод исключения Гаусса является наиболее широко используемым алгоритмом этого класса.

Явные итерационные методы. Наиболее известным алгоритмом этого класса является метод релаксации Якоби. Алгоритм выполняет следующие итерационные вычисления:

$$x_{i,j}^{new} = (x_{i-1,j}^{old} + x_{i,j-1}^{old} + x_{i+1,j}^{old} + x_{i,j+1}^{old}) / 4$$

Неявные итерационные методы. К этому классу относится метод последовательной верхней релаксации. Итерационное приближение вычисляется по формуле:

$$x_{i,j}^{new} = (w / 4) * (x_{i-1,j}^{new} + x_{i,j-1}^{new} + x_{i+1,j}^{old} + x_{i,j+1}^{old}) + (1-w) * x_{i,j}^{old}$$

Алгоритм Якоби

```
/* Jacobi-1 program */
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define  Max(a,b) ((a)>(b)?(a):(b))
#define  L  2000
#define  ITMAX 50
int i,j,it,k;
double eps;
double MAXEPS = 0.5;
double A[L][L], B[L][L];
int main(int an, char **as)
{
printf("JAC STARTED\n");
for(i=0; i<=L-1; i++)
  for(j=0; j<=L-1; j++)
  {
    A[i][j]=0.;
    B[i][j]=1.+i+j;

  }
for(it=1; it<=ITMAX; it++)
 {
   eps= 0.;
   for(i=1; i<=L-2; i++)
    for(j=1; j<=L-2; j++)
    {
      eps = Max(fabs(B[i][j]-A[i][j]),eps);
      A[i][j] = B[i][j];
    }
    for(i=1; i<=L-2; i++)
     for(j=1; j<=L-2; j++)
       B[i][j] = (A[i-1][j]+A[i+1][j]+
            A[i][j-1]+A[i][j+1])/4.;
    printf( "it=%4i   eps=%f\n", it,eps);
    if (eps < MAXEPS) break;
 }
 return 0;
}
```

Алгоритм последовательной верхней релаксации

```
/* Sor program */
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define  Max(a,b) ((a)>(b)?(a):(b))
#define  L  2000
#define  ITMAX 50
   int i,j,it,k;
   double eps;
   FILE *f;
   double A[L][L];
   double tmp;
int main(int an, char **as)
{
printf("SOR STARTED\n");
for(i=0; i<=L-1; i++)
  for(j=0; j<=L-1; j++)
  {
    if(i==0 || j==0 || i==L-1 || j==L-1) A[i][j]=0.;
    else A[i][j]=1.+i+j;
  }
/******  iteration loop  ************************/
for(it=1; it<=ITMAX; it++)
  {
    eps= 0.;
    for(i=1; i<=L-2; i++)
       for(j=1; j<=L-2; j++)
       {
          tmp=(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4.;
          eps = Max(fabs(tmp-A[i][j]),eps);
          A[i][j] = tmp;
       }
      printf( "it=%4i   eps=%f\n", it,eps);
  }
  f=fopen("sor.dat","wb");
  fwrite(A,sizeof(double),L*L,f);
  return 0;
}
```

Распределение циклов с зависимостью по данным

```
for(int i = 1; i < 100; i++)
  a[i]= a[i-1] + a[i+1]
```

Между витками цикла с индексами *i1* и *i2 (i1<i2)* существует зависимость по данным (информационная связь) массива *A,* если оба эти витка осуществляют обращение к одному элементу массива по схеме запись-чтение или чтение-запись.

Если виток *i1* записывает значение, а виток *i2* читает это значение, то между этими витками существует *потоковая зависимость* или просто зависимость *i1 -> i2.*

Если виток *i1* читает “старое” значение, а виток *i2* записывает “новое” значение, то между этими витками существует *обратная зависимость* *i1<- i2.*

В обоих случаях виток *i2* может выполняться только после витка *i1.*

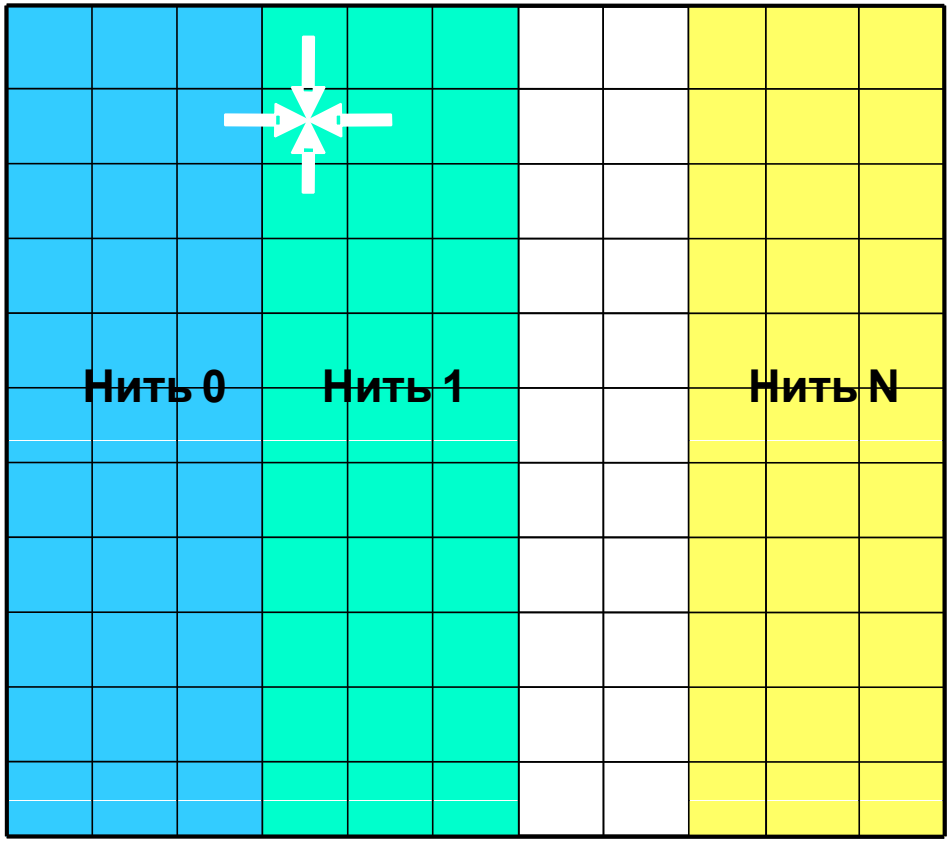
Распределение циклов с зависимостью по данным. Конструкция ordered

Цикл с зависимостью по данным может быть распараллелен с помощью конструкции ordered:

```
#pragma omp parallel for ordered
for(int i = 1; i < 100; i++) {
    #pragma omp ordered
      {
        a[i]= a[i-1] + a[i+1]
      }
}
```

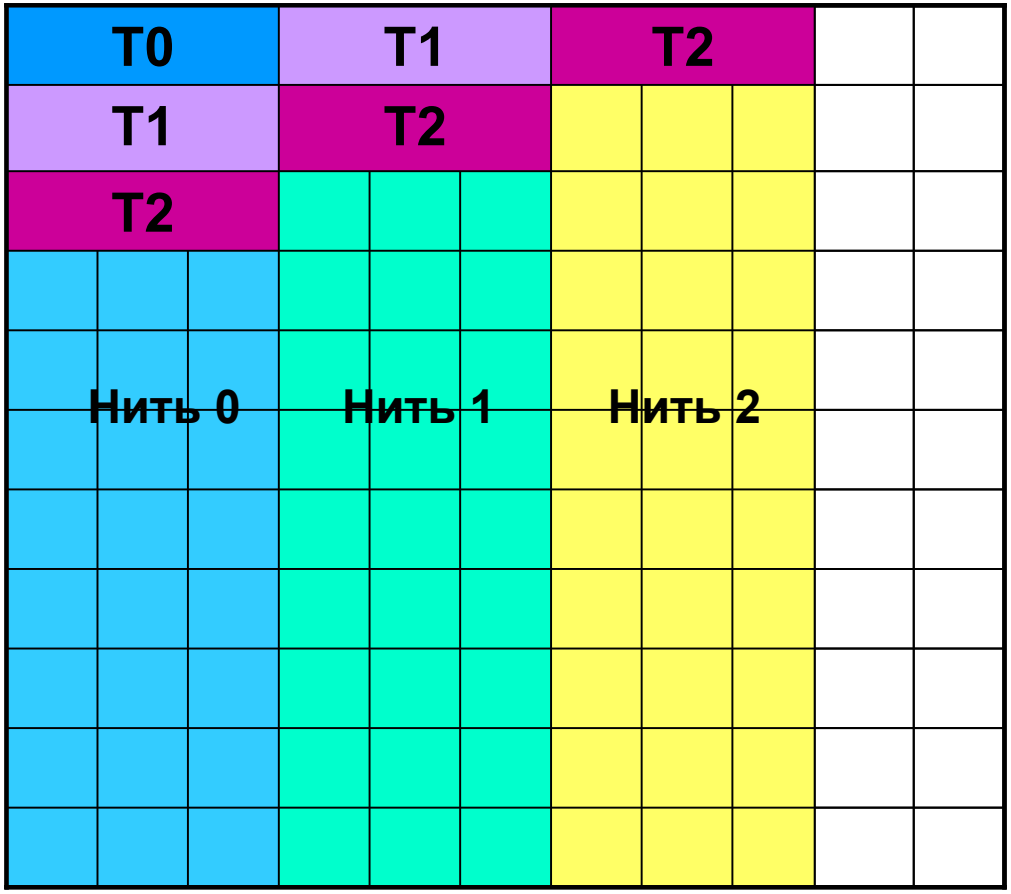
Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла.

```
for(int i = 1; i < N; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4
```



Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла.

```
for(int i = 1; i < N; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4
```



Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла.

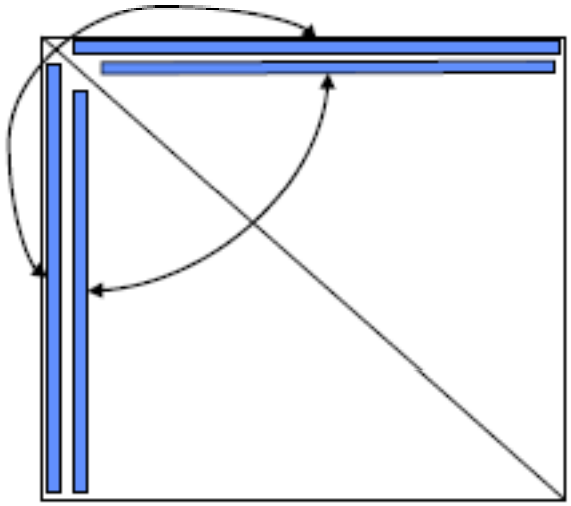
```
  int isync[NUMBER_OF_THREADS];
  int iam, numt, limit;
#pragma omp parallel private(iam,numt,limit)
  {
  iam = omp_get_thread_num ();
  numt = omp_get_num_threads ();
  limit=min(numt-1,N-2);
  isync[iam]=0;
#pragma omp barrier
  for (int i=1; i<N; i++) {
    if ((iam>0) && (iam<=limit)) {
      for (;isync[iam-1]==0;) {
        #pragma omp flush (isync)
      }
      isync[iam-1]=0;
      #pragma omp flush (isync)
    }
#pragma omp for schedule(static) nowait
    for (int j=1; j<N; j++) {
      a[i][j]=(a[i-1][j] + a[i][j-1] + a[i+1][j] +
               a[i][j+1])/4;
    }
    if (iam<ilimit) {
      for (;isync[iam]==1;) {
        #pragma omp flush (isync)
      }
      isync[iam]=1;
      #pragma omp flush (isync)
    }
  }
}
```

Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла.

```
#include <omp.h>
int main () {
  #pragma omp parallel private(iam,numt)
  {
    iam = omp_get_thread_num ();
    numt = omp_get_num_threads ();
    for (int newi=1; newi<N + numt - 1; newi++) {
      int i = newi - iam;
      #pragma omp for
        for (int j=1; j<N; j++) {
          if (i >= 1) && (i< N)) {
            a[i][j]=(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1])/4;
          }
        }
    }
  }
}
```

Балансировка нагрузки нитей.

```
#pragma omp parallel for private(tmp) shared (a)
    for (int i=0; i<N-2; i++)
        for (int j = i+1; j< N-1; j++) {
            tmp = a[i][j];
            a[i][j]=a[j][i];
            a[j][i]=tmp;
        }
```



Распределение витков цикла. Клауза schedule

Клауза schedule:
schedule(*алгоритм планирования[, число_итераций]*)

Где *алгоритм планирования* один из:
schedule(static*[, число_итераций]*) - статическое планирование;
schedule(dynamic*[, число_итераций]*) - динамическое планирование;
schedule(guided*[, число_итераций]*) - управляемое планирование;
schedule(runtime) - планирование в период выполнения;
schedule(auto) - автоматическое планирование (OpenMP 3.0).

```
#pragma omp parallel for schedule(static)
  for(int i = 1; i <= 100; i++)
     A[i]=0.;
```

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(static, 10)
  for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

- Поток 0 получает право на выполнение итераций 1-10, 41-50, 81-90.
- Поток 1 получает право на выполнение итераций 11-20, 51-60, 91-100.
- Поток 2 получает право на выполнение итераций 21-30, 61-70.
- Поток 3 получает право на выполнение итераций 31-40, 71-80

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(dynamic, 15)
  for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

- Поток 0 получает право на выполнение итераций 1-15.
- Поток 1 получает право на выполнение итераций 16-30.
- Поток 2 получает право на выполнение итераций 31-45.
- Поток 3 получает право на выполнение итераций 46-60.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 61-75.
- Поток 2 завершает выполнение итераций.
- Поток 2 получает право на выполнение итераций 76-90.
- Поток 0 завершает выполнение итераций.
- Поток 0 получает право на выполнение итераций 91-100.

Распределение витков цикла. Клауза schedule

число_выполняемых_потоком_итераций =
max(число_нераспределенных_итераций/omp_get_num_threads(),
число_итераций)

```
#pragma omp parallel for schedule(guided, 10)
  for(int i = 1; i <= 100; i++)
```

Пусть программа запущена на 4-х ядерном процессоре.

- Поток 0 получает право на выполнение итераций 1-25.
- Поток 1 получает право на выполнение итераций 26-44.
- Поток 2 получает право на выполнение итераций 45-59.
- Поток 3 получает право на выполнение итераций 60-69.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 70-79.
- Поток 2 завершает выполнение итераций.
- Поток 2 получает право на выполнение итераций 80-89.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 90-99.
- Поток 1 завершает выполнение итераций.
- Поток 1 получает право на выполнение 100 итерации.

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(runtime)
  for(int i = 1; i <= 100; i++)  /* способ распределения витков цикла между
    нитями будет задан во время выполнения программы*/
```

При помощи переменных среды:

csh:

```
setenv OMP_SCHEDULE "dynamic,4“
```

ksh:

```
export OMP_SCHEDULE="static,10“
```

Windows:

```
set OMP_SCHEDULE=auto
```

или при помощи функции системы поддержки:

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

Распределение витков цикла. Клауза schedule

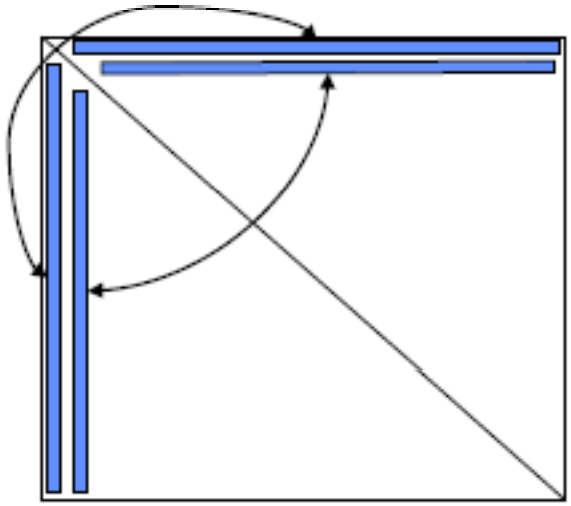
```
#pragma omp parallel for schedule(auto)
  for(int i = 1; i <= 100; i++)
```

Способ распределения витков цикла между нитями определяется реализацией компилятора.

На этапе компиляции программы или во время ее выполнения определяется оптимальный способ распределения.

Балансировка нагрузки нитей. Клауза schedule

```
#pragma omp parallel for private(tmp) shared (a) schedule (runtime)
    for (int i=0; i<N-2; i++)
        for (int j = i+1; j< N-1; j++) {
            tmp = a[i][j];
            a[i][j]=a[j][i];
            a[j][i]=tmp;
        }
```



```
export OMP_SCHEDULE="static"
export OMP_SCHEDULE="static,10"
export OMP_SCHEDULE="dynamic"
export OMP_SCHEDULE="dynamic,10"
```

Литература

- **OpenMP Application Program Interface Version 3.0, May 2008.** http://www.openmp.org/mp-documents/spec30.pdf
- **Антонов А.С.** Параллельное программирование с использованием технологии OpenMP: Учебное пособие.-М.: Изд-во МГУ, 2009. http://parallel.ru/info/parallel/openmp/OpenMP.pdf
- **Дж. Ортега.** Введение в параллельные и векторные методы решения линейных систем. – М.: Мир, 1991.
- **Воеводин В.В., Воеводин Вл.В.** Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.