КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Параллельное программирование для систем с общей памятью с использованием технологии OpenMP

Лабораторный практикум

Учебно-методическое пособие

КАЗАНСКИЙ УНИВЕРСИТЕТ
2016

УДК 681.3.06
ББК 018.2*32.973
П46

Печатается по рекомендации учебно-методической комиссии Института математики и механики им. Н.И. Лобачевского Казанского (Приволжского) федерального университета протокол № 2 от 20 октября 2016 года

Составители:
канд. физ.-мат. наук **О.А. Саченков,**
канд. физ.-мат. наук, доц. **Д.В. Бережной,**
канд. физ.-мат. наук **И.С. Балафендиева**

Рецензент –
канд. физ.-мат. наук, доц. **С.А. Кузнецов,**
канд. физ.-мат. наук, доц. **А.И. Еникеев**

Параллельное программирование для систем с общей памятью с использованием технологии OpenMP: Лабораторный практикум: учеб.-метод. пособие / О.А. Саченков, Д.В. Бережной, И.С. Балафендиева. – Казань: Казан. ун-т, 2016. – 26 с.

Данное пособие предназначено для проведения лабораторных и практических занятий по курсам программирования с применением технологии распараллеливания вычислений, в частности для курса «Теория и практика многопроцессорных вычислений в механике деформируемого твердого тела». Представлено краткое описание стандарта Open MP. Рассмотрены алгоритмы и их реализация на языке C/C++, такие как: распараллеливания умножения матрицы на вектор и решения систем линейных алгебраических уравнений.

УДК 681.3.06
ББК 018.2*32.973

© Саченков О.А., Бережной Д.В., Балафендиева И.С., 2016
© Казанский университет, 2016

СОДЕРЖАНИЕ

ОПИСАНИЕ СТАНДАРТА OPEN MP

В последнее время активно развивается подход к разработке параллельных программ, когда указания программиста по организации параллельных вычислений добавляются в программу при помощи тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный текст программы остается неизменным, и по нему, в случае отсутствия препроцессора, компилятор построит исходный последовательный программный код. В противном случае директивы параллелизма будут заменены на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки). Рассмотренный выше подход является основой технологии OpenMP, наиболее широко применяемой в настоящее время для организации параллельных вычислений на многопроцессорных системах с общей памятью. В рамках данной технологии директивы параллелизма используются для выделения в программе параллельных фрагментов, в которых последовательно исполняемый код может быть разделен на несколько раздельных командных потоков (threads). Далее эти потоки могут исполняться на разных процессорах (процессорных ядрах) вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (однопотоковых) и параллельных (многопотоковых) участков программного кода (см. рис. 1). Подобный принцип организации параллелизма получил наименование «вилочного» (fork-join) или пульсирующего параллелизма. Более полная информация по технологии OpenMP может быть получена в специальной литературе или в информационных ресурсах сети Интернет. Первый стандарт, определяющий технологию OpenMP применительно к языку Fortran, был принят в 1997 г., для алгоритмического языка C – в 1998 г. По-

следняя версия стандарта OpenMP для языков C и Fortran была опубликована в 2005 г.

Плюсы указанной технологии:

- Расширение последовательных языков путем введения параллельных конструкций в язык – директив OpenMP.
- Если компилятор не распознает OpenMP директиву, программа сохраняет функциональность.
- Базируется на идее нитевого программирования для систем с общей памятью.
- Включает в свой состав: параллельные конструкции, библиотеку функций, а также специальные переменные окружения.
- Индустриальный стандарт – Поддерживается Intel, Microsoft, Sun, IBM, HP и т.д. В ряде случаев поведение зависит от реализации.

Минусы указанной технологии:

- Переносимость только для систем с общей памятью.
- Масштабируемость в пределах одного узла.
- Возможная проблема размещения данных.
- Отсутствие возможности задать порядок нитей.

В общем виде программа представляется в виде набора последовательных (однопотоковых) и параллельных (многопотоковых) участков программного кода (см. рис. 1).

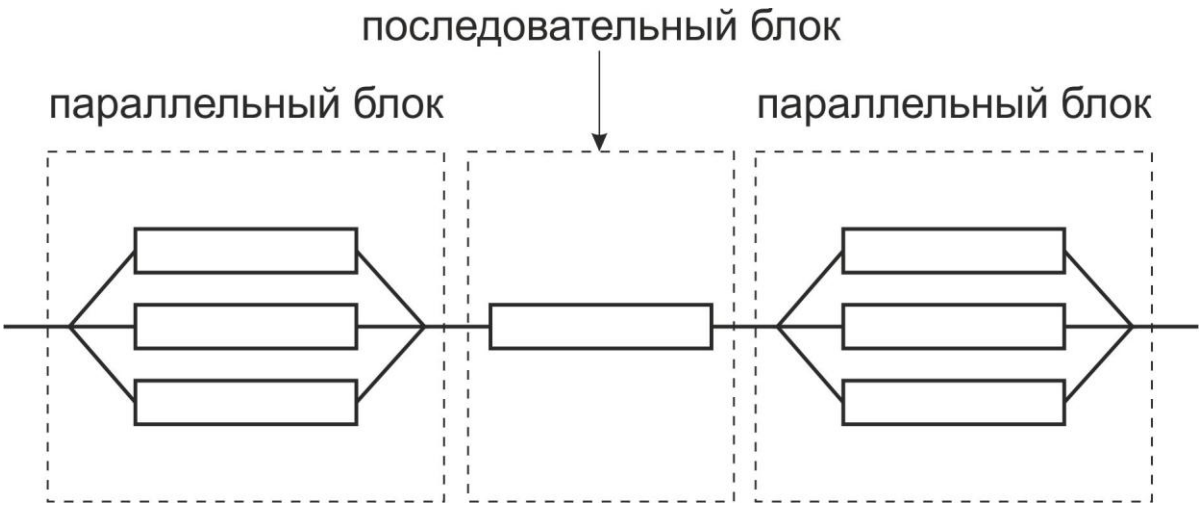

Рис. 1. Схема работы программы с параллельными областями.

Для использования технологии OMP в C/C++ необходимо подключить библиотеку omp:

```
#include <omp.h>
```

и при компилировании установить необходимый параметр компилятора. Если параметр */openmp* не указан, компилятор игнорирует предложения и директивы OpenMP. Вызовы функций OpenMP обрабатываются компилятором даже в том случае, если параметр */openmp* не указан. Указанный параметр можно включить через интерфейс проекта Visual C. Для этого откройте диалоговое окно *Страницы свойств проекта*, выберите *Проект→Свойства*, затем разверните узел *Свойства конфигурации*, разверните узел C/C++, выберите страницу свойств *Язык*, измените значение свойства *Поддержка OpenMP*.

Важно отметить, что разделение вычислений между потоками осуществляется под управлением соответствующих директив OpenMP. Равномерное распределение вычислительной нагрузки – балансировка (load balancing) – имеет принципиальное значение для получения максимально возможного ускорения выполнения параллельной программы. Потоки могут выполняться на разных процессорах (процессорных ядрах) либо могут группироваться для исполнения на одном вычислительном элементе (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор –

как правило, такой способ применяется для начальной проверки правильности параллельной программы. Количество потоков определяется в начале выполнения параллельных фрагментов программы и обычно совпадает с количеством имеющихся вычислительных элементов в системе; изменение количества создаваемых потоков может быть выполнено при помощи целого ряда средств OpenMP. Все потоки в параллельных фрагментах программы последовательно перенумерованы от 0 до n–1, где n есть общее количество потоков. Номер потока также может быть получен при помощи функции OpenMP.

Конструктивно в составе технологии OpenMP можно выделить: директивы, библиотеку функций, набор переменных окружения.

Модель разделяемой памяти имеет следующие особенности: нити взаимодействуют через разделяемые переменные; разделение определяется синтаксически; любая переменная, видимая более чем одной нитью, является разделяемой (*shared*); любая переменная, видимая только одной нитью, является *private.*

Возможны Race conditions: требуется синхронизация для предотвращения конфликтов; возможно управление доступом (*shared*, *private*) для минимизации накладных расходов на синхронизацию.

Далее по тексту алгоритмы будут реализованы на языке C/C++, код в тексте будет выделен `шрифтом Courier New`, ***курсивом*** будут отмечены задания для самостоятельной работы. Приведенные результаты были реализованы на процессоре Intel Core i3-3210 CPU @ 3.20GHz.

РАБОТА №1. ДИРЕКТИВЫ ПАРАЛЛЕЛЬНЫХ ОБЛАСТЕЙ OMP

1. Определение директив OMP

Рассмотрим подробнее синтаксис директив OpenMP.

```
#pragma omp directive_name [clause[clause ...]] newline
<структурный блок >
```

Действия, соответствующие директиве, применяются непосредственно к структурному блоку, расположенному за директивой. Структурным блоком может быть любой оператор, имеющий единственный вход и единственный выход.

Количество нитей, выполняющих работу, определяется переменной окружения *OMP_NUM_THREADS или* вызовом функции *omp_set_num_threads().*

Определение «своих» координат в вычислительном пространстве:

- свой номер: *omp_get_thread_num()*;

- число нитей: *omp_get_threads_num()*.

Как уже отмечалось выше, мастер-нить имеет номер 0.

2. Определение параллельной области

Рассмотрим основную директиву параллельной области. Стандартный формат директивы parallel представляет собой:

```
#pragma omp parallel [clause ...] newline
<структурный блок >
```

Рассмотрим ряд возможных параметров *clause*:

- *if (scalar_expression)* - если условие в *if* не выполняется, то процессы не создаются;
- *private (list)* - определяет список переменных, которые будут локальными для каждого потока; переменные создаются в момент формирования потоков параллельной области; начальное значение переменных является неопределенным;
- *shared (list)* - определяет список переменных, которые будут общими для всех потоков параллельной области; правильность использования таких переменных должна обеспечиваться программистом;

- *reduction (operator: list)* - определяет оператор и переменную для редуцирования;
- *num_threads(integer-expression)* - определяет количество потоков.

Рассмотрим подробнее управление областью видимости данных. Как было отмечено выше, для переменных есть два типа видимости: *shared и private*. Ниже приведены примеры объявления переменной *j shared и private*.

```
#pragma omp parallel shared(j)
#pragma omp parallel private(j)
```

По умолчанию переменные, видимые в области, объемлющей блок параллельного исполнения, являются общими (shared). Переменные, объявленные внутри блока, по умолчанию считаются закрытыми (private). На рис. 2 проиллюстрировано различие для указанной переменной *j* в зависимости от ее области видимости.

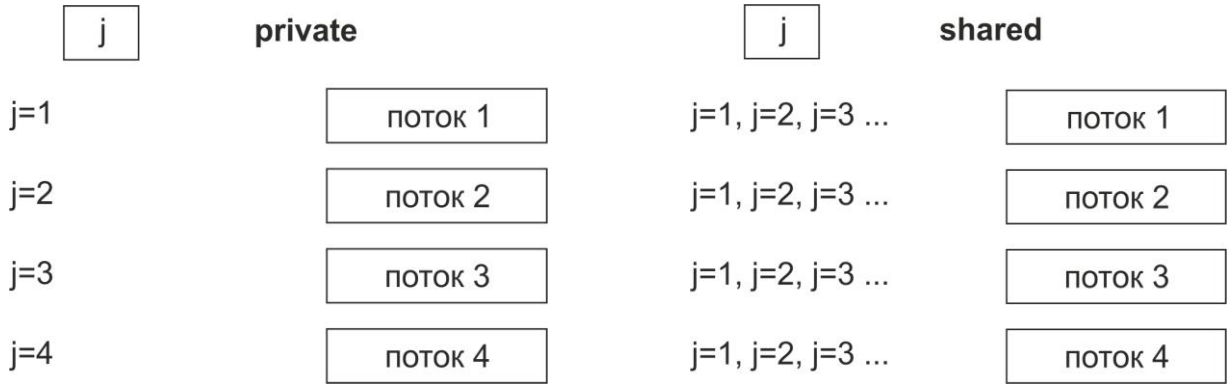

Рис. 2. Иллюстрация к видимости переменных.

Рассмотрим классический пример и выведем на экран фразу «Hello World» из различных нитей.

```
#include <omp.h>
main () {
 int nthreads, tid;
 #pragma omp parallel private(tid) // Создание парал-
лельной области
 {
   tid = omp_get_thread_num();
   printf("Hello World from thread = %d\n", tid); // пе-
чать номера потока
   if (tid == 0) {
     nthreads = omp_get_num_threads();
```

```
     printf("Number of threads = %d\n", nthreads); //
Печать (master) количества потоков
   }
  } // Завершение параллельной области
}
```

В этом случае на экран будет выведено сообщение (см. рис. 3), при этом *Number of threads* должно соответствовать количеству процессов на компьютере; порядок вывода по процессам в общем случае произвольный.

```
Hello World from thread = 0
Hello World from thread = 2
Hello World from thread = 1
Hello World from thread = 3
Number of threads = 4
Для продолжения нажмите любую клавишу . . .
```

Рис. 3. Результат работы программы.

Для определения количества потоков вызовите диспетчер задач (*Ctrl+Alt+Del* для WinOs) и выберите закладку *Быстродействие* (в различных версиях и локализациях WinOs название закладки может отличаться).

Рассмотрим на примере операцию редуцирования. В этом случае необходимо задать операцию и переменные редуцирования. Перед выполнением параллельной области для каждого потока создаются копии этих переменных; потоки формируют значения в своих локальных переменных; при завершении параллельной области над всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных. Рассмотрим работу директивы на примере суммирования двух массивов:

```
#include <omp.h>
   main () { // vector dot product
     int i, n, chunk;
     float a[NMAX], b[NMAX], result;
     n = NMAX; chunk = CHUNK;
     result = 0.0;
     for (i=0; i < n; i++) {
       a[i] = i * 1.0; b[i] = i * 1.0; }
      #pragma omp parallel for default(shared) \
```

```
      schedule(static,chunk) reduction(+:result)
        for (i=0; i < n; i++)
          result = result + (a[i] * b[i]);
        printf("Final result= %f\n",result);
   system("pause");
  return 0;
}
```

3. Директива распараллеливания циклов

Рассмотрим директиву для распараллеливания циклов:

```
#pragma omp for [clause ...] newline
for loop
```

Возможные параметры (*clause*): *private(list)*; *firstprivate(list); lastprivate(list); reduction(operator: list); ordered; schedule(kind[, chunk_size]);nowait.*

Распределение итераций в директиве *for* регулируется параметром (*clause*) *schedule*:

- *static* – итерации делятся на блоки по *chunk* итераций и статически разделяются между потоками; если параметр *chunk* не определен, итерации делятся между потоками равномерно и непрерывно;
- *dynamic* – распределение итерационных блоков осуществляется динамически (по умолчанию *chunk*=1);
- *guided* – размер итерационного блока уменьшается по экспоненциальному закону при каждом распределении;
- *chunk* определяет минимальный размер блока (по умолчанию *chunk*=1);
- *runtime* – правило распределения определяется переменной *OMP_SCHEDULE* (при использовании *runtime* параметр *chunk* задаваться не должен).

Рассмотрим работу директивы на примере суммирования двух массивов:

```
#include <omp.h>
#define CHUNK 100
```

```
#define NMAX 1000
main()
{
 int i, n, chunk, tid;
 float a[NMAX], b[NMAX], c[NMAX];
  for (i=0; i < NMAX; i++){
  a[i] = i*1.0; b[i] = i*1.0; };
  n = NMAX; chunk = CHUNK;
  #pragma omp parallel shared(a,b,c,n,chunk) private(i)
     #pragma omp for schedule(dynamic,chunk) nowait
       for (i=0; i < n; i++) {
         c[i] = a[i] + b[i];
         tid = omp_get_thread_num();
         printf("counted from %d\n", tid);
       }
  // end of parallel section
 system("pause");
 return 0;
}
```

Задание: *увеличьте размерность массива в примере, посмотрите распределение вычислений по потокам, установите* shedule *значение* static, *сравните результаты.*

4. Директива параллельных секций

Для выделения раздельных фрагментов кода в параллельные области используется директива *sections.* Отметим ряд свойств этой директивы: каждый фрагмент выполняется однократно (директива *section*); разные фрагменты выполняются разными потоками; завершение директивы по умолчанию синхронизируется; директивы *sections* должны использоваться только в статическом контексте. Рассмотрим вышеуказанный пример с применением директивы *sections*. В этом случае цикл разделяется на две половины, и каждый цикл считается на разных потоках.

```
#include <omp.h>
#define NMAX 1000
main () {
 int i, n, tid;
  float a[NMAX], b[NMAX], c[NMAX];
  for (i=0; i < NMAX; i++)
    a[i] = b[i] = i * 1.0;
```

```
  n = NMAX;
  #pragma omp parallel shared(a,b,c,n) private(i)
  {
      #pragma omp sections nowait
  {
         #pragma omp section
            for (i=0; i < n/2; i++) {
 c[i] = a[i] + b[i]; tid = omp_get_thread_num();
printf("counted from %d\n", tid); }
         #pragma omp section
            for (i=n/2; i < n; i++) {
 c[i] = a[i] + b[i]; tid = omp_get_thread_num();
printf("counted from%d\n", tid); }
      }    // end of sections
  }    // end of parallel section
 system("pause");
 return 0;}
```

Задание: *разделите циклы на три или четыре части, посмотрите результаты.*

Задание: *Реализуйте алгоритм численного подсчета определенного интеграла с помощью директив reduction, section, parallel for.*

5. Директива непараллельных блоков

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз (например, запись в файл), то следует использовать директиву *single*.

```
#pragma omp single [name] newline
<структурный блок >
```

Одна нить будет выполнять данный фрагмент, а все остальные нити будут ожидать завершения её работы, если только не указана опция *nowait*.

С помощью директив *critical* оформляется критическая секция программы, которая запрещает одновременное исполнение структурированного блока более чем одним потоком.

```
#pragma omp critical [name] newline
<структурный блок >
```

В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение дан-

ной критической секции. Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

РАБОТА №2. РЕАЛИЗАЦИЯ АЛГОРИТМОВ АЛГЕБРЫ МАТРИЦ С ПРИМЕНЕНИЕМ ТЕХНОЛОГИИ OMP

Классической областью применения распараллеливания является алгебра. Решение больших систем линейных алгебраических уравнений, действия с матрицами и векторами занимают много машинного времени и при этом хорошо распараллеливаются. Существует два основных принципа распараллеливания: ленточное и блочное разбиение. При ленточном разбиении каждому потоку выделяется подмножество строк или столбцов матрицы, или же происходит чередование строк или столбцов. При блочном разбиении матрица делится на прямоугольные наборы элементов.

1. Умножение матрицы на вектор

Рассмотрим умножение матрицы на вектор. Формула для подсчета компонент вектора имеет вид:

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, \; 0 \le i < m \qquad (2.1)$$

Получение результирующего вектора **c** предполагает повторение m операций по умножению строк матрицы **A** на вектора **b**. Каждая операция включает перемножение элементов строки матрицы **A** и вектора **b** (n операций) и последующее суммирование полученных произведений (n-1 операций). Общее количество необходимых скалярных операций T = m·(2n-1). Для упрощения будем предполагать, что матрица **A** квадратная.

Рассмотрим алгоритм умножения матрицы на вектор при разложении по строкам. В этом случае перемножение каждой строки матрицы на вектор выполняется на отдельном процессе (см. рис. 4).

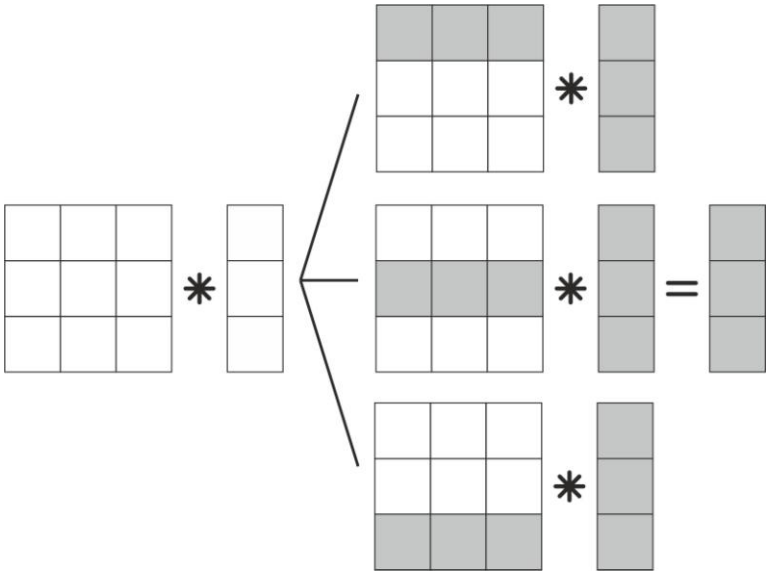

Рис. 4. Иллюстрация алгоритма умножение матрицы
на вектор при разложении по строкам.

Рассмотрим реализацию этого алгоритма. Определим основную функцию:

```
void main () {
 double* pMatrix;
 double* pVector;
 double* pResult;
 int Size;
 // Создание матрицы и векторов
 ProcessInit (pMatrix, pVector, pResult, Size);
 // Перемножение матрицы на вектор
 SerialProduct (pMatrix, pVector, pResult, Size);
// Параллельное перемножение матрицы на вектор
 ParallelProduct (pMatrix, pVector, pResult, Size);
//  Выгрузка и завершение
 ProcessTerminate (pMatrix, pVector, pResult, Size);
}
```

Задание: *Реализуйте функции ProcessInit, SerialProduct, ProcessTerminate. Для функций SerialProduct, ParallelProduct реализуйте вывод затраченного времени в файл/на экран.*

Реализация распараллеливания умножения матрицы на вектор по строкам приведена ниже:

```
void ParallelProduct (double*  pMatrix, double* pVector,
double* pResult, int Size)  {
```

```
 int i, j;
 #pragma omp parallel for private (j)
 for (i=0; i< Size; i++) {
  for (j=0; j< Size; j++) {
    pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
 }
}
```

Здесь переменная *j* обозначена как *private*, это необходимо для того чтобы каждый процесс обладал своим индексом суммирования при перемножении строки матрицы на вектор.

Рассмотрим алгоритм умножения матрицы на вектор при разложении по столбцам. В этом случае происходит перемножение каждого столбца матрицы на компоненту вектора на разных процессах, после чего производится суммирование этих произведений (см. рис. 5).

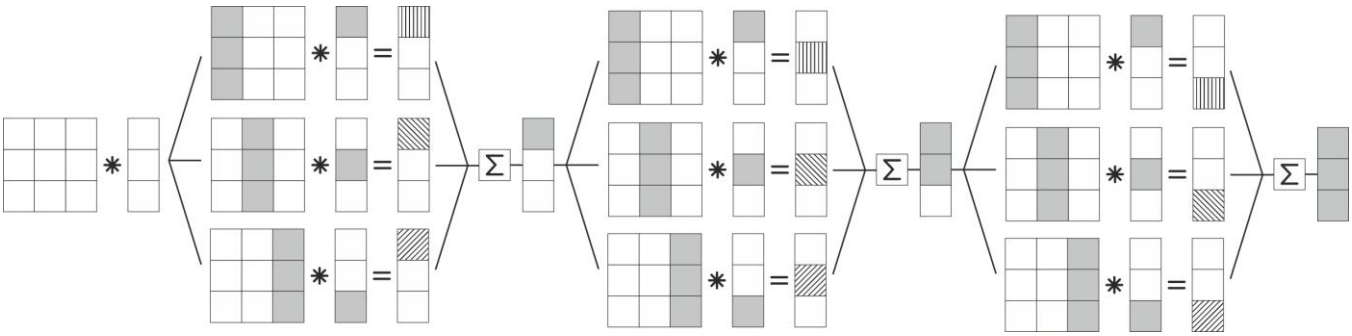

Рис. 5. Иллюстрация алгоритма умножение матрицы на вектор при разложении по столбцам.

Рассмотрим реализацию этого алгоритма:

```
void ParallelProduct C(double*  pMatrix, double*
pVector, double* pResult, int Size)  {
   int i, j;
  double IterSum =0;
  for (i=0; i< Size; i++) {
    IterSum =0;
    #pragma omp parallel for reduction (+:IterSum)
    for (j=0; j< Size; j++)
      IterSum += pMatrix[i*Size+j]*pVector[j];
    pResult[i] = IterSum;
 }
}
```

Здесь в цикле по индексу *j* задана операция редукции по операции суммирования «+» для переменной *IterSum*, которая хранит слагаемых компоненты вектора.

Рассмотрим теперь блочное разбиение. При блочном разбиении матрица делится на прямоугольные наборы элементов. Для упрощения в дальнейших рассуждениях будем считать блоки квадратными (см. рис. 6). Для эффективного выполнения параллелизма целесообразно выделить число параллельных потоков, совпадающих с количеством блоков матрицы A. Если количество потоков равно p, то размерность блока равна $q=p^{1/2}$, а количество строк и столбцов в блоке: k=m/q, l=n/q.

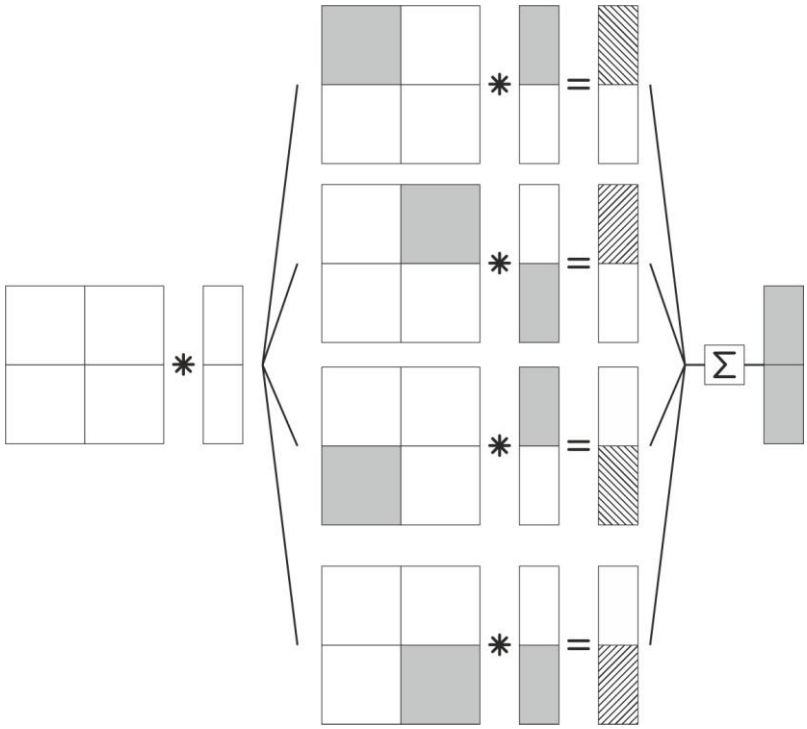

Рис. 6. Иллюстрация алгоритма умножения матрицы
на вектор при блочном разбиении.

Рассмотрим реализацию алгоритма перемножения матрицы на вектор при блочном разбиении:

```
void ParallelProductB (double*  pMatrix, double*
pVector, double* pResult, int Size)  { // Параллельная
реализация - блочное разложение
 int ThreadID;
 int GridThreadsNum = 4;
```

```
 int GridSize = int(sqrt(double(GridThreadsNum)));
 int BlockSize = Size/GridSize;
 omp_set_num_threads(GridThreadsNum);
#pragma omp parallel private(ThreadID)
 {
   ThreadID=omp_get_thread_num();
   double* pThreadResult = new double[Size];
   for (int i=0;i<Size;i++)
    pThreadResult[i]=0;
int i_start = (int(ThreadID/GridSize))*BlockSize;
 int j_start = (ThreadID%GridSize)*BlockSize;
 double IterResult;
for (int i=0;i<BlockSize;i++){
    IterResult = 0;
    for (int j=0;j<BlockSize;j++)
     IterResult +=
pMatrix[(i+i_start)*Size+(j+j_start)]*pVector[j+i_start]
;
   pThreadResult[i+i_start] = IterResult;
   }
   #pragma omp critical
   for (int i=0;i<Size;i++)
    pResult[i] += pThreadResult[i];
   delete [] pThreadResult;
 }
}
```

В переменную *GridThreadsNum* задается количество потоков, которые будут задействованы, размер блоков *BlockSize* рассчитывается по количеству потоков. В переменной *ThreadID (private)* храниться номер актуального потока. На основании этого значения рассчитываются смещения для переменных *i, j*, по которым заданы циклы.

Проведем численные эксперименты для оценки результативности рассмотренных алгоритмов. На рис. 7 приведены результаты этих экспериментов (для удобства на рисунке отмечены Serial - перемножение без параллелизма, P.Rows - разложение по строкам, P.Columns - разложение по столбцам, P.Blocks - разложение по блокам).

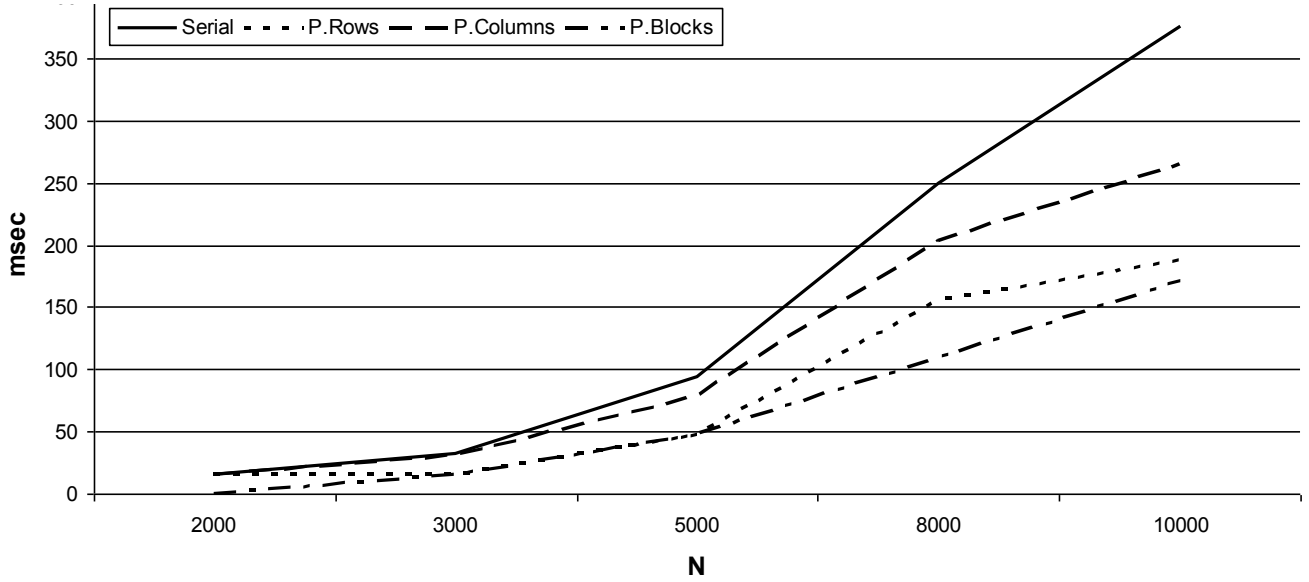

Рис. 7. Результаты численного моделирования, рассмотренных алгоритмов.

Стоит отметить, что приведенные результаты будут различны на разных ЭВМ. На основании приведенных данных можно заключить, что наиболее выигрышным является метод распараллеливания по блокам.

Задание: *Реализуйте функции рассмотренных алгоритмов и проведите численные эксперименты, организуйте запись затраченного времени в файл, постройте графики, проанализируйте их.*

Задание: *Реализуйте аналогичные алгоритмы для случая перемножения двух матриц, проведите численные эксперименты, организуйте запись затраченного времени в файл, постройте графики, проанализируйте их.*

РАБОТА №3. РЕАЛИЗАЦИЯ АЛГОРИТМОВ РЕШЕНИЯ СЛАУ С ПРИМЕНЕНИЕМ ТЕХНОЛОГИИ OMP

Рассмотрим применение технологии распараллеливания для решения систем линейных алгебраических уравнений (СЛАУ). Для примера рассмотрим метод Якоби. Дана СЛАУ вида:

$$Ax = b \tag{3.1}$$

Построим матрицы D, B такие что, A=D-B

$$D = diag(A) = \begin{pmatrix} a_{11} & 0 & ... & 0 \\ 0 & a_{22} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & a_{nn} \end{pmatrix}, B = D - A = \begin{pmatrix} 0 & -a_{12} & ... & -a_{1n} \\ -a_{21} & 0 & ... & -a_{2n} \\ ... & ... & ... & ... \\ -a_{n1} & -a_{n2} & ... & 0 \end{pmatrix} \tag{3.2}$$

Тогда систему (3.1) можно привести к виду:

$$x = D^{-1}Bx + D^{-1}b \tag{3.3}$$

Будем рассматривать левую часть как актуальную итерацию, тогда можно перейти к итерационной формуле:

$$x^{k+1} = D^{-1}Bx^{k} + D^{-1}b \tag{3.4}$$

Условие сходимости будет:

$$\| Ax^{k} - b \| < \varepsilon \tag{3.5}$$

Распишем итерационную формулу (3.4)

$$\begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}^{k+1} = \begin{pmatrix} 0 & -\frac{a_{12}}{a_{11}} & ... & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & ... & -\frac{a_{2n}}{a_{22}} \\ ... & ... & ... & ... \\ -\frac{a_{n1}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & ... & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}^{k} + \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ ... \\ \frac{b_n}{a_{nn}} \end{pmatrix} = \begin{pmatrix} \frac{b_1 - a_{12}x_2 ... - a_{1n}x_n}{a_{11}} \\ \frac{b_2 - a_{21}x_1 ... - a_{2n}x_n}{a_{22}} \\ ... \\ \frac{b_n - a_{n1}x_1 - a_{n2}x_2 ... - 0}{a_{nn}} \end{pmatrix}^{k} \tag{3.6}$$

Несложно показать, что *i*-ая компонента вектора актуальной итерации определяется через предыдущую по формуле:

$$x_i^{k+1} = \frac{b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^{k}}{a_{ii}} \tag{3.7}$$

Из конструкции формулы видно, что выгодно производить расчет итерации каждой компоненты вектора решений на отдельном потоке. Приведем блок-схему алгоритма (рис. 8).

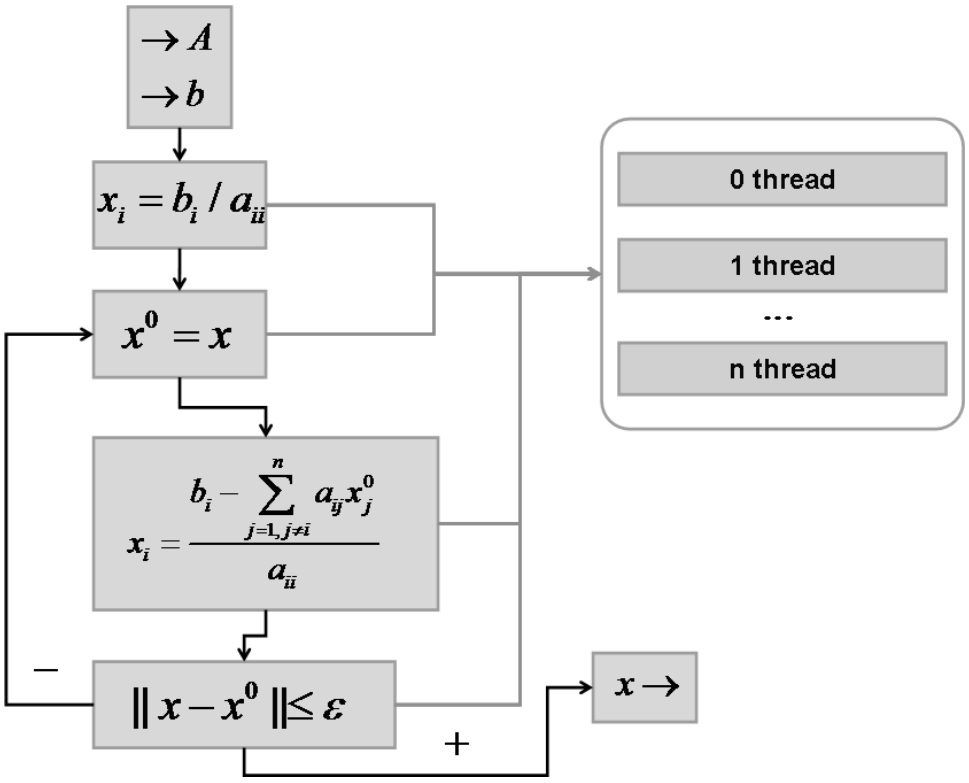

Рис. 8. Блок схема алгоритма параллельной реализации метода Якоби.

Из блок-схемы легко видеть, что существует несколько операций, позволяющих распараллеливание: операция присвоения вектора, итерационная формула Якоби и подсчет нормы вектора.

Рассмотрим подробнее реализацию распараллеливания итерационной формулы Якоби:

```
void Solve(void)
{
    double* X = new double[Dimension];
    double* X_last = new double[Dimension];
    for (int I=0; I<Dimension; I++)
        X_last[I] = B[I]/A[I*Dimension + I];
    const double Epsilon = 0.0001;
    double maxDifference;
    int IterationCounter = 1;
    do
    {
        if (IterationCounter > 1)
```

```
            this->CopyArray(X, X_last);
        #pragma omp parallel for
        for (int I = 0; I < Dimension; I++)
        {
            X[I] = B[I];
            for (int J = 0; J < Dimension; J++)
            {
                if (I != J)
                    X[I] -= A[I*Dimension + J] *
X_last[J];
            }
            X[I] = X[I] / A[I*Dimension + I];
        }
        maxDifference = MaxDifference(X, X_last);
        IterationCounter++;
    } while (maxDifference > Epsilon);
}
```

Здесь распараллеливание реализовано для итераций компонент вектора решений. На рис. 9 приведена относительная разница во времени расчетов между последовательной и параллельной реализацией при различных размерностях системы, для полученных результатов приведена аппроксимация параболой четвертой степени (пунктирная кривая).

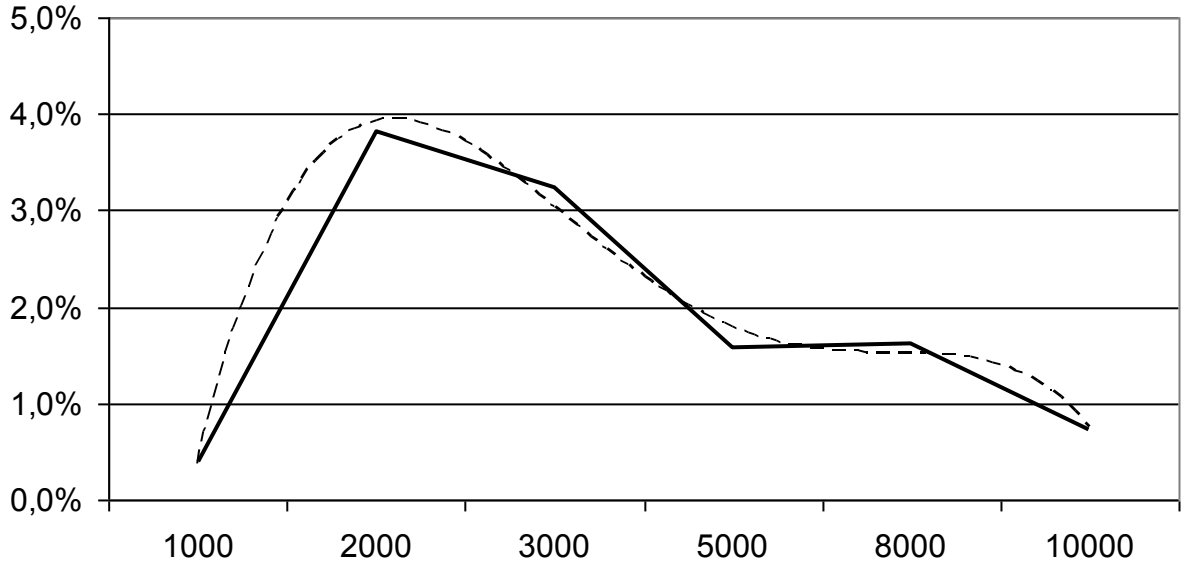

Рис. 9. Результаты численного моделирования, рассмотренного алгоритма.

Задание: *Реализуйте распараллеливание оставшихся блоков итерационного метода Якоби согласно блок схеме на рис. 8. Проведите расчеты и сравните результаты с последовательной реализаций.*

Задание: *Реализуйте распараллеливание итерационного метода Зейделя и проведите численные эксперименты для методов Зейделя и Якоби, организуйте запись затраченного времени расчета в файл, постройте графики, проанализируйте их.*

ЗАДАНИЯ

Задания 1

1. Дано число n, на i-ой нити определите n^i, затем найдите сумму всех полученных чисел.

2. Дано число n, на i-ой нити определите n·(i+1), затем найдите произведение всех полученных чисел.

3. В матрице *A*(*m*,*n*) увеличьте каждую строку кратную i в (i+1) раз, где i - номер нити.

4. В матрице *A*(*m*,*n*) сложите все строки кратные i, где i - номер нити.

5. В матрице *A*(*m*,*n*) сложите все столбцы кратные i, где i - номер нити.

Задания 2

1. В матрице *A*(*m*,*n*) найдите наименьший и наибольший элементы.

2. Найдите сумму всех строк матрицы *A*(*m*,*n*) и умножьте ее на сумму всех столбцов.

3. Для матрицы *A*(*m*,*n*) определите все матрицы алгебраических дополнений.

4. Матрицу *M*(*k*, *n*) заполнить следующим способом: элементам, находящимся на периферии (по периметру матрицы) присвоить значение 1; периметру оставшейся подматрицы – значение 2, элементам следующего окаймления – значение 3, и так далее до заполнения всей матрицы.

5. Каждый элемент вектора *A*(*n*) (кроме двух крайних) заменить выражением $a_i' = (a_{i-1} + 2a_i + a_{i+1})/4$, *i*=2,3,…*n*-1, а крайние элементы – выражениями: $a_1' = (a_1 + a_2)/2$; $a_n' = (a_{n-1} + a_n)/2$.

Задания 3

1. Реализуйте подсчет определенного интеграла на интервале [a,b] для функции *f(x)* методом левых и правых прямоугольников.

2. Реализуйте подсчет определенного интеграла на интервале [a,b] для функции *f(x)* методом центральных прямоугольников.

3. Реализуйте подсчет определенного интеграла на интервале [a,b] для функции *f(x)* методом трапеций.

4. Реализуйте подсчет определенного интеграла на интервале [a,b] для функции *f(x)* методом Симпсона.

5. Реализуйте подсчет определенного интеграла на интервале [a,b] для функции *f(x)* методом Гаусса.

Задания 4

Реализуйте последовательное и параллельное вычисления следующих выражений (заглавными буквами обозначены матрицы, маленькими - вектора):

1	$b\cdot(A\cdot b^T + d^T)$	5	$b\cdot(A-C)-d$	9	$b\cdot(d-b\cdot C)$
2	$A\cdot b^T + (b\cdot C)^T$	6	$b\cdot A - d\cdot C$	10	$b\cdot(A-C)-d\cdot A$
3	$(d-b\cdot C)\cdot d^T$	7	$A\cdot b^{T\ T} - d\cdot C$	11	$(d\cdot A - b\cdot A)\cdot b^T$
4	$b\cdot(A-C)\cdot b^T$	8	$(A\cdot b^T - d^T)\cdot b^T$	12	$b\cdot(A-C)-d\cdot C$

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. The OpenMP Architecture Review Board http://openmp.org/wp/

2. OpenMP Application Program Interface, Version 3.0. May, 2008. – 318 p.

3. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: *Учебное пособие*. – М.: Изд-во МГУ, 2009. – 77 с.

4. Gebali, Fayez. Algorithms and parallel computing. A John Wiley & Sons, Inc., Publication, 2011. – 365 p.