

Министерство образования и науки Российской Федерации

Московский физико – технический институт

(Государственный университет)

Кафедра радиоэлектроники и прикладной информатики

**Эффективность и ускорение
параллельных программ**

Лабораторная работа № 2

по курсу

Параллельное программирование

Москва 2011

УДК 004.424

Лабораторная работа №2 по курсу : Параллельное программирование.

Составители:

Пальян Р.Л., Гаврилов Д.А., Леус А.В., Филимонов А.В.

Лабораторная работа составлена при поддержке компании Intel.

Московский физико-технический институт
Кафедра радиоэлектроники и прикладной информатики
141700, Моск. обл, г. Долгопрудный, Институтский пер. 9
(С) Московский физико-технический институт, 2011

Параллельная программа – это множество взаимодействующих параллельных процессов. Основной целью параллельных вычислений является ускорение решения вычислительных задач. Параллельные программы обладают следующими особенностями:

- 1) осуществляется управление работой множества процессов;
- 2) организуется обмен данными между процессами;
- 3) утрачивается детерминизм поведения из-за асинхронности доступа к данным;
- 4) преобладают нелокальные и динамические ошибки;
- 5) появляется возможность тупиковых ситуаций;
- 6) возникают проблемы масштабируемости программы и балансировки загрузки вычислительных узлов.

Рассмотрим некоторый последовательный алгоритм решения какой-либо задачи. В нем есть как операции, которые не могут выполняться параллельно (например, ввод/вывод), так и операции, которые можно выполнять на нескольких процессорах одновременно. Пусть доля последовательных операций в алгоритме равна α .

Время выполнения последовательного алгоритма обозначим T_1 . Время выполнения параллельной версии алгоритма на p одинаковых процессорах можно записать следующим образом:

$$T_p = \alpha T_1 + \frac{(1-\alpha)T_1}{p}. \quad (1)$$

Ускорением параллельного алгоритма называют отношение времени выполнения лучшего последовательного алгоритма к времени выполнения параллельного алгоритма:

$$S = \frac{T_1}{T_p}. \quad (2)$$

Параллельный алгоритм может давать большое ускорение, но использовать для этого множество процессов неэффективно. Для оценки масштабируемости параллельного алгоритма используется понятие эффективности:

$$E = \frac{S}{p}. \quad (3)$$

Теоретическую оценку максимального ускорения, достижимого для параллельного алгоритма с долей последовательных операций равной α определяет закон Амдала:

$$S = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} \leq \frac{1}{\alpha}. \quad (4)$$

Таким образом, если всего 10% операций алгоритма не может быть выполнена параллельно, то никакая параллельная реализация данного алгоритма не может дать больше ускорение более чем в 10 раз.

Основные понятия технологии MPI

В рамках данной работы изучается параллельное программирование для систем с распределенной памятью. В системах этого типа на каждом вычислительном узле работают процессы, реализующие некоторый параллельный алгоритм. У каждого процесса есть своя собственная область памяти, к которой ни один другой процесс не имеет доступа. Все взаимодействия осуществляются с помощью передачи сообщений между процессами.

Существует множество способов организации передачи сообщений. В данной работе рассматривается разработка параллельных программ с помощью технологии MPI.

MPI – программный интерфейс для передачи сообщений между процессами. Стандартизацией MPI занимается организация MPI Forum. Реализации интерфейса MPI существуют для множества различных платформ.

В рамках технологии MPI на каждом вычислительном узле запускается копия параллельной программы. Каждая копия получает ранг – уникальный идентификатор, использующийся для адресации сообщений.

Обмен сообщениями в рамках MPI происходит между процессами, которые относятся к одному коммунитатору. Коммунитатор – это способ группировки процессов. По умолчанию все запущенные процессы попадают в коммунитатор `MPI_COMM_WORLD`.

MPI предусматривает несколько вариантов обмена сообщениями. Сообщения бывают типа «точка-точка» – между двумя процессами, и «коллективные» – между несколькими процессами одновременно.

Также отправка и прием сообщений бывают блокирующим и неблокирующим (асинхронными). В случае блокирующего обмена передающие и принимающие процессы блокируются до тех пор, пока передача сообщения не завершится. Например, в случае блокирующего обмена типа «точка-точка» отправитель будет приостановлен до тех пор, пока получатель не вызовет функцию получения сообщения и получит его.

Структура программы и основные функции MPI

Структура программы, использующей MPI, выглядит следующим образом:

1. Подключение библиотеки MPI.
2. Инициализация среды MPI.
3. Работа программы, обмен сообщениями.
4. Остановка среды MPI.

Для подключения библиотеки MPI в программе на языке C нужно включить заголовочный файл `mpi.h`.

Для инициализации среды MPI на каждом вычислительном узле необходимо один и только один раз вызвать функцию `MPI_Init(int* argc, char*** argv)`. Все прочие MPI-процедуры могут быть вызваны только после вызова `MPI_Init`.

`MPI_Init` получает адреса аргументов, стандартно получаемых самой `main` от операционной системы и хранящих параметры командной строки. В конец командной

строки программы MPI-загрузчик `mpirun` добавляет ряд информационных параметров, которые требуются `MPI_Init`.

После успешной инициализации каждый процесс может узнать свой ранг с помощью `MPI_Comm_rank` (`MPI_Comm comm, int* rank`). Также можно узнать общее число процессов в коммуникаторе, вызвав `MPI_Comm_size` (`MPI_Comm comm, int* size`). В качестве аргументов обеим функциям передается идентификатор коммуникатора (обычно, `MPI_COMM_WORLD`) и адрес переменной, куда будет записано требуемое значение.

Остановка среды MPI осуществляется вызовом функции `MPI_Finalize`(). Все последующие обращения к любым MPI-процедурам, в том числе к `MPI_Init`, запрещены. К моменту вызова `MPI_Finalize` некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Для обмена сообщениями типа «точка-точка» используется следующий набор функций:

- ♣ `MPI_Send(void* buffer, int count, MPI_Type type, int dst, int tag, MPI_Comm comm)` – блокирующая отправка.
- ♣ `MPI_Isend(void* buffer, int count, MPI_Type type, int dst, int tag, MPI_Comm comm, MPI_Request* request)` – неблокирующая отправка.
- ♣ `MPI_Recv(void* buffer, int count, MPI_Type type, int src, int tag, MPI_Comm comm, MPI_Status* status)` – блокирующий прием.
- ♣ `MPI_Irecv(void* buffer, int count, MPI_Type type, int src, int tag, MPI_Comm comm, MPI_Request* request)` – блокирующий прием.

Параметры всех этих функции очень похожи:

- ♣ `buffer` – указатель на начало области памяти, откуда будут передаваться (или куда будут приниматься данные).
- ♣ `count` – число элементов в буфере.
- ♣ `type` – тип элемента в буфере. В MPI поддерживаются все основные типы языка C, а также можно создавать произвольные пользовательские типы элементов.
- ♣ `dst/src` – ранг принимающего/передающего процесса.
- ♣ `tag` – метка сообщения. Служит для выделения логического типа сообщений.
- ♣ `comm` – коммуникатор, в рамках которого будет вестись обмен (обычно `MPI_COMM_WORLD`).
- ♣ `status` – указатель на структуру, в которой будет информация о статусе доставки сообщения. Если статус не интересует, то можно передать специальное значение `MPI_STATUS_IGNORE`.
- ♣ `request` – указатель на структуру, в которой будет информация о статусе передачи сообщения.

Для точного измерения времени работы библиотека MPI предоставляет функцию `MPI_Wtime`(). Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Для измерения времени работы

фрагмента кода используется следующая конструкция:

```
double t_start = MPI_Wtime();  
// некоторый код  
double t_end = MPI_Wtime();  
double working_time = t_end - t_start;
```

Для компиляции и запуска программы, использующей MPI, на компьютере должна быть установлена и настроена реализация библиотеки MPI (например, OpenMPI или MPICH2).

Компиляция программы осуществляется с помощью команды:

```
mpicc -o <название_исполняемого_файла> <имя_исходного_файла>.c
```

Запуск осуществляется с помощью команды:

```
mpirun -n <число_запускаемых_процессов> <название_исполняемого_файла>  
[аргументы]
```

Условия Бернштейна

Пусть в программе имеются два оператора S1 и S2, непосредственно динамически следующих друг за другом. Пусть W(S) – набор выходных переменных оператора S, а R(S) – набор его входных переменных. Тогда возможность их одновременного выполнения различными исполнителями в параллельной системе можно определить с помощью условий Бернштейна.

Если для операторов S1 и S2, непосредственно динамически следующих друг за другом, выполнено:

- ▲ пересечение W(S1) и W(S2) пусто;
- ▲ пересечение W(S1) и R(S2) пусто;
- ▲ пересечение R(S1) и W(S2) пусто;

то они могут быть исполнены параллельно.

Анализ зависимостей в простых циклах

Во многих программах, связанных с математическим моделированием, приходится практически одинаковым образом обрабатывать большие массивы данных. Так что особый интерес представляет анализ существующих последовательных программ на параллелизм по данным. Распараллеливание по данным предполагает разделение массивов на зоны, каждая из которых обрабатывается отдельным исполнителем, — так называемые зоны ответственности исполнителей. Подобные вычисления обычно реализуются в последовательном коде с помощью операторов цикла. В работе рассматривается способ определения наличия зависимостей по данным для циклов, работающих с массивами, и влияние этих зависимостей на возможность параллельного выполнения циклов.

Пусть тело цикла состоит из двух операторов S1 и S2, в наборы входных и/или

выходных данных которых входит обращение к элементам одного и того же одномерного массива данных A .

```
for (int i=0; i<i_fin; ++i) {  
  S1: ....(A[f(i)])...  
  S2: ...A[g(i)]...  
}
```

Здесь $f(i)$ и $g(i)$ — некоторые целочисленные функции целого переменного. Для простоты будем считать, что индекс массива A может принимать любое целое значение. Нашей основной задачей является выяснение того, можно ли разбить итерационное пространство такого цикла — целочисленный отрезок $[1, j_{fin}]$ — на зоны ответственности для параллельного выполнения. Вспомним, что на самом деле оператор цикла — это просто форма сокращения исходного текста программы. Если убрать это сокращение и развернуть цикл, то получим:

```
S1,1: ....(A[f(1)])...  
S2,1: ...A[g(1)]...  
S1,2: ....(A[f(2)])...  
S2,2: ...A[g(2)]...  
...  
S1,i_fin: ....(A[f(i_fin)])...  
S2,i_fin: ...A[g(i_fin)]...
```

В такой развернутой последовательности следующих друг за другом операторов можно провести анализ их совокупности на зависимость по данным. Будем полагать, что для одного оператора в теле цикла обращение к элементу массива A входит в набор входных переменных, а для другого — в набор выходных элементов.

Без ограничения общности получаем цикл:

```
for (int i=0; i<i_fin; ++i) {  
  S1: A[f(i)] = ....  
  S2: .... = ...A[g(i)]...  
}
```

Легко видеть, что условия Бернштейна могут быть нарушены в том случае, если существуют значения итерационной переменной « i » λ и κ , $1 \leq \lambda \leq i_{fin}$, $1 \leq \kappa \leq i_{fin}$ такие, что $f(\lambda) = g(\kappa)$. Чтобы узнать существуют ли такие значения, нужно решить приведенное уравнение при указанных ограничениях в целых числах. В общем случае определить, имеет ли уравнение решения, и найти их алгоритмически невозможно.

В простых случаях, например, когда f и g — линейные функции, определить, существует ли решение, и каково оно, конечно, возможно, но в общем случае — нет. Если решения не существует, то все операторы развернутого цикла независимы друг от друга и могут быть выполнены одновременно различными исполнителями, скажем, каждая

итерация цикла — на своем исполнителе.

Пусть решение существует, и мы нашли соответствующие κ и λ . Условия Бернштейна нарушены — между операторами есть зависимость. В этом случае оператор S_1 (где элемент массива A — выходная переменная) называют источником (source) зависимости, а оператор S_2 (где элемент массива A — входная переменная) называют стоком (sink) зависимости. Вычислим величину $D = \lambda - \kappa$ (из итерации стока вычитаем итерацию источника). Эту величину принято называть расстоянием зависимости цикла.

Расстояние зависимости играет важную роль при анализе цикла на параллельность. Его значение позволяет определять тип возникающей зависимости по данным и возможность разбиения итерационного пространства на зоны ответственности для параллельного исполнения.

Если расстояние зависимости $D < 0$, то между операторами тела цикла существует антизависимость. Цикл может быть распараллелен так, что каждая итерация будет выполняться отдельным исполнителем, если перед началом выполнения итераций продублировать необходимые входные данные на исполнителях.

Если расстояние зависимости $D > 0$, то между операторами тела цикла существует потоковая зависимость. При $D > 1$ цикл может быть распараллелен не более чем на D исполнителях.

Если расстояние зависимости $D = 0$, то тип зависимости между операторами тела цикла в общем случае не определен. Цикл может быть распараллелен так, что каждая итерация будет выполняться отдельным исполнителем.

Анализ зависимостей во вложенных циклах

В современных научных исследованиях часто рассматриваются задачи, имеющие более одного измерения. При построении математических моделей таких задач приходится использовать многомерные массивы данных, а для их обработки в программах, реализующих построенные модели, — применять вложенные циклы. Нам необходимо уметь применять к подобным программным конструкциям аппарат анализа зависимостей по данным для возможного распараллеливания последовательного кода.

Рассмотрим нормализованный цикл:

```
for (int j1=0; j1<u1; ++j1) {  
    for (int j2=0; j2<u2; ++j2) {  
        ....  
        for (int jn=0; jn<un; ++jn) {  
            }  
        }  
    }  
}
```

В таких циклах конкретная итерация определяется совокупностью значений всех счетчиков j_1, j_2, \dots, j_n . Будем рассматривать их набор как n -мерный вектор $J = (j_1, j_2, \dots, j_n)$ и назовем его итерационным вектором. Множество всех допустимых значений итерационных векторов образует итерационное пространство цикла. В этом пространстве

между векторами можно ввести отношения порядка. Будем говорить, что $I = J$, если $\forall k, 1 \leq k \leq n, i_k = j_k$, и что $I < J$ в том случае, когда $\exists s, 1 \leq s \leq n$, такое что $\forall k, 1 \leq k < s, i_k = j_k$, а $i_s < j_s$. Как и в случае с одномерным циклом предположим, что тело цикла состоит из двух операторов $S1$ и $S2$, в наборы входных и/или выходных данных которых входит обращение к элементам одного и того же массива данных A с размерностью, совпадающей с количеством уровней вложенности цикла. Пусть индексы массива могут принимать произвольные целочисленные значения. При этом для простоты допустим, что для оператора $S1$ элемент массива A принадлежит к выходным переменным оператора, а для оператора $S2$ — к входным переменным. Тогда можно представить цикл в виде:

```

for (int j1=0; j1<j_fin1; ++j1) {
    for (int j2=0; j2<j_fin2; ++j2) {
        ....
        for (int jn=0; jn<i_finn; ++jn) {
            S1: A[f1(J),...,fn(J)] = ....
            S2: .... = ... A[g1(J),...,gn(J)] ...
        }
    }
}

```

Здесь функции $f_k(J)$ и $g_k(J)$, $1 \leq k \leq n$, есть целочисленные функции от n целых переменных. Задачей является выяснение возможности разбиения итерационного пространства такого цикла на зоны ответственности для параллельного выполнения.

Понятно, что условия Бернштейна нарушаются. Зависимость возникает, если имеет решение система уравнений $F(K) = G(\Lambda)$, где F — вектор-функция (f_1, \dots, f_n) , а G — вектор-функция (g_1, \dots, g_n) .

Введем для цикла понятие вектора расстояний зависимости (или просто вектора расстояний) следующим образом: $D = \Lambda - K$ (из вектора итераций, соответствующего итерации стока зависимости, вычитаем вектор итерации, соответствующий итерации источника зависимости).

Определить тип существующей зависимости по данным и возможность распараллеливания цикла по виду вектора расстояний не так просто. Поэтому вводится понятие вектора направлений для цикла. Компоненты вектора направлений d (а это — символьный вектор) определяются следующим образом:

- ▲ $d_i = „=“$, $D_i = 0$;
- ▲ $d_i = „>“$, $D_i < 0$;
- ▲ $d_i = „<“$, $D_i > 0$.

Если многомерный цикл имеет вектор направлений $d = („=“, \dots, „=“)$, то цикл может быть распараллелен по произвольному количеству индексов без всяких ограничений. При этом циклы, соответствующие различным уровням вложенности первоначальной конструкции, можно безопасно менять местами.

Пусть многомерный цикл имеет вектор направлений d , в состав которого входят только элементы «>» и «=». Такой цикл может быть распараллелен без всяких ограничений по любому количеству индексов, соответствующих компонентам «=» в векторе направлений. Распараллеливание по индексам, соответствующим компонентам «>» в векторе направлений, возможно при дублировании необходимых входных данных. Перед распараллеливанием циклы, соответствующие различным уровням вложенности первоначальной конструкции, можно безопасно менять местами.

Пусть многомерный цикл имеет вектор направлений d , в состав которого входят только элементы „<“ и „=“. Такой цикл может быть распараллелен без всяких ограничений по любому количеству индексов, соответствующих компонентам „=“ в векторе направлений. Распараллеливание по индексам, соответствующим компонентам „<“ в векторе направлений, проблематично. Перед распараллеливанием циклы, соответствующие различным уровням вложенности первоначальной конструкции, можно безопасно менять местами.

Пусть для некоторого многомерного цикла определен вектор направлений d . Истинная зависимость в цикле существует тогда и только тогда, когда крайний левый элемент вектора направлений, отличный от „=“, есть „<“.

Для произвольного цикла возможно распараллеливание по любому индексу, соответствующему компоненту „=“ в векторе направлений. Уровень вложенности, соответствующий этому компоненту, можно поменять местами с любым соседним уровнем вложенности с сохранением результата вычислений. Два соседних уровня вложенности, которым соответствуют одинаковые компоненты вектора направлений, также можно поменять местами. Если в цикле существует антизависимость, то распараллеливание возможно по произвольному количеству индексов при дублировании необходимых входных данных. Распараллеливание для циклов с истинной зависимостью может быть проблематично.

Естественно, что для цикла, в котором зависимости возникают по элементам не одного, а нескольких массивов, решение о возможности распараллеливания принимается по результатам анализа всей совокупности зависимостей.

Практическая часть

Основным заданием данной лабораторной работы является разработка и исследование параллельной программы на основе существующей готовой последовательной. Полученные результаты требуется сравнить, а так же изобразить графически для каждой из реализаций зависимость коэффициента ускорения программы от количества используемых исполнителей.

Лабораторная работа подразумевает выполнение распараллеливания представленной ниже программы, а так же дополнительно индивидуальную задачу для каждого студента, полученную из данной последовательной:

```
#include <stdio.h>

#include <stdlib.h>

#define ISIZE 1000
```

```

#define JSIZE 1000

int main(int argc, char **argv)
{
    double a[ISIZE][JSIZE];
    int i, j;
    FILE *ff;
    for (i=0; i<ISIZE; i++){
        for (j=0; j<JSIZE; j++){
            a[i][j] = 10*i +j;
        }
    }
    for (i=0; i<ISIZE; i++){
        for (j = 0; j < JSIZE; j++){
            a[i][j] = sin(0.00001*a[i][j]);
        }
    }
    ff = fopen("result.txt","w");
    for(i=0; i < ISIZE; i++){
        for (j=0; j < JSIZE; j++){
            fprintf(ff,"%f",a[i][j]);
        }
        fprintf(ff,"\n");
    }
    fclose(ff);
}

```

Предлагается на выбор 16 последовательных программ:

Задача 1а int main(int argc, char **argv) {	Задача 1б int main(int argc, char **argv) {
---	---

<pre> double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=1; i<ISIZE; i++){ for (j = 0; j < JSIZE-1; j++){ a[i][j] = sin(0.00001*a[i- 1][j+1]); } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } } fprintf(ff,"\n"); } fclose(ff); } </pre>	<pre> double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=0; i<ISIZE-3; i++){ for (j = 4; j < JSIZE; j++){ a[i][j] = sin(0.00001*a[i+3][j-4]); } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } } fprintf(ff,"\n"); } fclose(ff); } </pre>
<p>Задача 1в</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=2; i<ISIZE; i++){ for (j = 0; j < JSIZE-3; j++){ a[i][j] = sin(0.00001*a[i- 2][j+3]); } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } } } </pre>	<p>Задача 1г</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=1; i<ISIZE; i++){ for (j = 3; j < JSIZE-1; j++){ a[i][j] = sin(0.00001*a[i- 1][j-3]); } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } } } </pre>

<pre> fprintf(ff, "\n"); } fclose(ff); } </pre>	<pre> fprintf(ff, "\n"); } fclose(ff); } </pre>
<p>Задача 1д</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=1; i<ISIZE-1; i++){ for (j = 6; j < JSIZE-1; j++){ a[i][j] sin(0.00001*a[i+1][j-6]); } } ff = fopen("result.txt", "w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff, "%f ", a[i][j]); } fprintf(ff, "\n"); } fclose(ff); } </pre>	<p>Задача 2а</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=0; i<ISIZE-1; i++){ for (j = 1; j < JSIZE; j++){ a[i][j] sin(0.00001*a[i+1][j-1]); } } ff = fopen("result.txt", "w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff, "%f ", a[i][j]); } fprintf(ff, "\n"); } fclose(ff); } </pre>
<p>Задача 2б</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=0; i<ISIZE-3; i++){ for (j = 2; j < JSIZE; j++){ a[i][j] sin(0.00001*a[i+3][j-2]); } } } </pre>	<p>Задача 2в</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=3; i<ISIZE; i++){ for (j = 0; j < JSIZE-2; j++){ a[i][j] = sin(0.00001*a[i- 3][j+2]); } } } </pre>

<pre> } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>	<pre> } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>
<p>Задача 2г</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=0; i<ISIZE-4; i++){ for (j = 5; j < JSIZE; j++){ a[i][j] sin(0.00001*a[i+4][j-5]); } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>	<p>Задача 2д</p> <pre> int main(int argc, char **argv) { double a[ISIZE][JSIZE]; int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; } } for (i=8; i<ISIZE; i++){ for (j = 0; j < JSIZE-3; j++){ a[i][j] = sin(0.00001*a[i- 8][j+3]); } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f",a[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>
<p>Задача 3а</p> <pre> #include <stdio.h> #include <stdlib.h> #define ISIZE 100 #define JSIZE 100 double a[ISIZE][JSIZE], b[ISIZE][JSIZE]; int main(int argc, char **argv) { </pre>	<p>Задача 3б</p> <pre> #include <stdio.h> #include <stdlib.h> #define ISIZE 100 #define JSIZE 100 double a[ISIZE][JSIZE], b[ISIZE][JSIZE]; int main(int argc, char **argv) { </pre>

<pre> int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; b[i][j] = 0.; } } for (i=0; i<ISIZE; i++){ for (j = 0; j < JSIZE; j++){ a[i][j] sin(0.00001*a[i][j]); } } for (i=0; i<ISIZE-1; i++){ for (j = 0; j < JSIZE; j++){ b[i][j] = a[i+1][j]*1.5; } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f ",b[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>	<pre> int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; b[i][j] = 0.; } } for (i=0; i<ISIZE; i++){ for (j = 0; j < JSIZE; j++){ a[i][j] sin(0.00001*a[i][j]); } } for (i=0; i<ISIZE-1; i++){ for (j = 3; j < JSIZE; j++){ b[i][j] = a[i+1][j-3]*2; } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f ",b[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>
<p>Задача 3в</p> <pre> #include <stdio.h> #include <stdlib.h> #define ISIZE 100 #define JSIZE 100 double a[ISIZE][JSIZE], b[ISIZE][JSIZE]; int main(int argc, char **argv) { int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; b[i][j] = 0.; } } for (i=0; i<ISIZE; i++){ </pre>	<p>Задача 3г</p> <pre> #include <stdio.h> #include <stdlib.h> #define ISIZE 100 #define JSIZE 100 double a[ISIZE][JSIZE], b[ISIZE][JSIZE]; int main(int argc, char **argv) { int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; b[i][j] = 0.; } } for (i=0; i<ISIZE; i++){ </pre>

<pre> for (j = 0; j < JSIZE; j++){ a[i][j] } sin(0.00001*a[i][j]); } for (i=0; i<ISIZE; i++){ for (j = 2; j < JSIZE; j++){ b[i][j] = a[i][j-2]*2.5; } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f ",b[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>	<pre> for (j = 0; j < JSIZE; j++){ a[i][j] } sin(0.00001*a[i][j]); } for (i=5; i<ISIZE; i++){ for (j = 0; j < JSIZE-2; j++){ b[i][j] = a[i-5][j+2]*1.5; } } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f ",b[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>
<p>Задача 3д</p> <pre> #include <stdio.h> #include <stdlib.h> #define ISIZE 100 #define JSIZE 100 double a[ISIZE][JSIZE], b[ISIZE][JSIZE]; int main(int argc, char **argv) { int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; b[i][j] = 0.; } } for (i=0; i<ISIZE; i++){ for (j = 0; j < JSIZE; j++){ a[i][j] } sin(0.00001*a[i][j]); } for (i=0; i<ISIZE-3; i++){ for (j = 5; j < JSIZE; j++){ b[i][j] = a[i+3][j-5]*3; } } } </pre>	<p>Задача 3е</p> <pre> #include <stdio.h> #include <stdlib.h> #define ISIZE 100 #define JSIZE 100 double a[ISIZE][JSIZE], b[ISIZE][JSIZE]; int main(int argc, char **argv) { int i, j; FILE *ff; for (i=0; i<ISIZE; i++){ for (j=0; j<JSIZE; j++){ a[i][j] = 10*i +j; b[i][j] = 0.; } } for (i=0; i<ISIZE; i++){ for (j = 0; j < JSIZE; j++){ a[i][j] } sin(0.00001*a[i][j]); } for (i=0; i<ISIZE-4; i++){ for (j = 1; j < JSIZE; j++){ b[i][j] = a[i+4][j-1]*1.5; } } } </pre>

<pre> } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f ",b[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>	<pre> } ff = fopen("result.txt","w"); for(i=0; i < ISIZE; i++){ for (j=0; j < JSIZE; j++){ fprintf(ff,"%f ",b[i][j]); } fprintf(ff,"\n"); } fclose(ff); } </pre>
--	--

Задание к допуску:

1. Вычислить вектор направлений для вашего варианта задания.
2. Определить тип зависимости и возможные варианты распараллеливания.

Задание к выполнению:

Написать параллельную реализацию предложенной последовательной программы. В случае, когда распараллеливание возможно по нескольким индексам, выбрать наиболее эффективный вариант. Параллельная и последовательная реализации должны генерировать одинаковый выходной файл.

Задание к сдаче:

Построить график зависимости коэффициента ускорения от числа исполнителей.

Определить эффективность параллельной реализации.

Контрольные вопросы:

1. Ускорение и эффективность параллельных алгоритмов.
2. Закон Амдаля.
3. Свойства канала передачи данных. Латентность.
4. Виды обменов «точка-точка»: синхронные, асинхронные. Буферизация данных.
5. Синхронизация выполнения.
6. Условия Бернштейна.
7. Расстояние зависимости. Его влияние на возможность распараллеливания простого цикла.
8. Расстояние зависимости для вложенных циклов.
9. Вектор направлений. Его влияние на возможность распараллеливания вложенных циклов.
10. Условия возможности перестановки вложенных циклов с сохранением результата вычислений.
11. Проанализировать возможность распараллеливания следующих циклов:

- 1) `for (int i=0; i<N; ++i) { a[i] = d[i] + 5*i; c[i] = a[2*i] * 2;}`
- 2) `for (int i=0; i<N; ++i) { c[i] = sin(a[2*i]); a[i] = d[i]*2; }`
- 3) `for (int i=0; i<N; ++i) { a[i] = a[i-1]*2;}`
- 4) `for (int i=0; i<N; ++i) { a[i] = a[i+4]/2;}`
- 5) `for (int i=0; i<N; ++i) { a[i] = a[i+4]*tan(a[i-1]);}`

Приложение:

Официальная страница документации MPI Forum:

<http://www.mpi-forum.org/docs/docs.html>

Карпов В.Е. «Введение в распараллеливание алгоритмов и программ»

<http://crm.ics.org.ru/journal/article/1725/>