**Задание:**

**Вычислить прямое произведение множеств А1, А2, А3, А4. Входные данные: множества чисел А1, А2, А3, А4, мощности множеств могут быть не равны между собой и мощность каждого множества ≥ 1. (Прямое произведение множеств это не нахождение произведения всех элементов, это математический термин, предполагающий построение нового множества по определенным правилам.)**

**Лабораторная работа**

**Тема:** Программирование многопоточных приложений в Pthread.

**Цели:** Научиться разбивать алгоритм на подзадачи, для последующего их параллельного выполнения различными потоками. Получить навык распараллеливания алгоритмически сложной задачи.

**Порядок выполнения лабораторной работы**

1. Разработать алгоритм решения задания, с учетом разделения вычислений между несколькими потоками. Желательно избегать ситуаций изменения одних и тех же общих данных несколькими потоками. Если же избежать этого невозможно, необходимо использовать мьютексы, семафоры или блокировки чтения-записи. Об этих механизмах можно прочитать ниже. Составить схему потоков.
2. Реализовать алгоритм и протестировать на нескольких примерах.

**Пример №1**

Задание: найти результат произведения матриц А\*B, где А и B – матрицы 3х3.

 Разделим задачу между тремя потоками. Каждый поток будет вычислять элементы результирующей матрицы, стоящие в определенной строке.



Рис. 4. Схема потоков

 Реализуем задачу, используя библиотеку PTHREAD.

#include <stdio.h>

#include <pthread.h>

int A[3][3] = { 1, -2, 0, 4, 6, 2, -3, 4, -2 } ;

int B[3][3] = {0, 2, 0, 1, 1, 1, 5, -3, 10} ;

int C[3][3] ; *//результирующая матрица*

*//стартовая функция для дочерних потоков*

void\* func(void \*param)

{ int p=\*(int \*)param ;  *//номер строки, заполняемой потоком*

*// вычисление элементов результирующей матрицы,*

*//стоящих в строке с номером p*

for (int i=0 ; i<3 ; i++)

 { C[p][i]=0 ;

 for (int j=0 ; j<3 ; j++)

 C[p][i]+=A[p][j]\*B[j][i] ;

 }

}

int main()

{ *//идентификаторы для дочерних потоков*

pthread\_t mythread1, mythread2 ;

int num[3] ; for (int i=0 ; i<3 ; i++) num[i]=i ; *//номера строк для потоков*

*//создание дочерних потоков*

pthread\_create(&mythread1, NULL, func, (void \*)(num+1)) ;

pthread\_create(&mythread2, NULL, func, (void \*)(num+2)) ;

*//заполнение первой строки результирующей матрицы*

func((void \*)num) ;

*//ожидание завершения дочерних потоков*

pthread\_join(mythread1,NULL) ;

pthread\_join(mythread2,NULL) ;

*// вывод результата вычислений всех потоков*

for (int i=0 ; i<3 ; i++)

 { fprintf(stdout,"\n") ;

 for (int j=0 ; j<3 ; j++)

 fprintf(stdout, "%d ",C[i][j]) ;

 } return 1;

}

 В приведенном примере не используются мьютексы или подобные им средства синхронизации, т.к. каждый поток осуществляет запись исключительно в выделенные для него элементы массива, индексы которых не пересекаются для различных потоков.

 **Пример №2**

 Варианты заданий в лабораторной работе являются достаточно сложными алгоритмическими задачами, даже при их реализации в виде последовательного алгоритма. Многие варианты могут быть решены переборными алгоритмами, когда вычисляется множество всех возможных комбинаций и из него выделяются искомые варианты (не исключено, что отдельные комбинаторные задачи, могут быть решены более коротким способом с использованием других интеллектуальных решений).

 Рассмотрим следующую задачу комбинаторного перебора. Имеется три массива чисел произвольной размерности А, В, С. Необходимо составить новый массив D, каждый элемент которого является суммой всех возможных троек A[i]+B[j]+C[k].

 Рассмотрим частный случай решения задачи.

 A={ 1, 2, 3 }, B = { 4, 5 }, C = {6, 7, 8 }.

 Тогда D = { 1+4+6, 1+4+7, 1+4+8, 1+5+6, 1+5+7, 1+5+8,

 2+4+6, 2+4+7, 2+4+8, 2+5+6, 2+5+7, 2+5+8,

 3+4+6, 3+4+7, 3+4+8, 3+5+6, 3+5+7, 3+5+8 }

 Рамерность D составляет 18 элементов. Размерности исходных множеств А, В, С равны 3, 2, 3 элементам соответственно. 18 = 3 \* 2 \* 3.

 В более сложных случаях размерность множества переборных элементов может соотноситься с одним из специальных математических коэффициентов (см. в разделе "Теория вероятности и мат. статистика") Ank, Bnk, Cnk, Dnk.

 Реализуем последовательный алгоритм.

int A[3] ={ 1, 2, 3 } ; int NA = 3 ; *// NA - число элементов в А*

int B[2] ={ 4, 5 } ; int NB = 2 ; *// NB - число элементов в B*

int C[3] ={ 6, 7, 8 } ; int NC = 3 ; *// NC - число элементов в C*

int \*D ; D = new int [NA\*NB\*NC] ;

int i, j, k, l=0 ; *//i для перебора индексов в А, j в B, k в C, l в D*

for (i=0 ; i<NA ; i++) *//перебор всех элементов в А*

for (j=0 ; j<NB ; j++) *//перебор всех элементов в B*

for (k=0 ; k<NC ; k++) *//перебор всех элементов в C*

{ D[l]= A[i]+B[j]+C[k] ; l++ ; } *//помещение элемента в массив D*

 Разделим выполнение задачи между двумя потоками. Для этого достаточно разделить между ними итерации, например, первого цикла напополам. Тогда для указанных чисел первый поток найдет все комбинации сумм с 1 из массива А, а второй поток найдет комбинации сумм с 2 и 3 из массива А.

 Также потоки должны разделить между собой общий массив D, куда они будут записывать свои результаты. При этом предпочтительно разделить массив D так, чтобы потоки не соревновались за захват индекса массива и не могли обратиться к индексу потока-соседа. Представим, что работает только первый поток, тогда сколько всего элементов он вычислит? Т.к. он работает до середины первого массива А, и полностью перебирает элементы оставшихся массивов В и С, получаем, что он заполнит массив из [(NA/2)\*NB\*NC] элементов, где реально заполняются элементы с индексами 0, 1, ..., (NA/2)\*NB\*NC-1. Значит второй поток может заполнять массив D, начиная с индекса (NA/2)\*NB\*NC и заканчивая NA\*NB\*NC-1 включительно.

|  |
| --- |
|  *//Глобальные переменные*int A[3] ={ 1, 2, 3 } ; int NA = 3 ; *// NA - число элементов в А*int B[2] ={ 4, 5 } ; int NB = 2 ; *// NB - число элементов в B*int C[3] ={ 6, 7, 8 } ; int NC = 3 ; *// NC - число элементов в C*int \*D ; D = new int [NA\*NB\*NC] ;  |
| *Поток 1* | *Поток 2* |
| *//локальные переменные потока 1*int i, j, k, **l=0** ;for (i=**0** ; i<**NA/2** ; i++) for (j=0 ; j<NB ; j++) for (k=0 ; k<NC ; k++) { D[l]= A[i]+B[j]+C[k] ; l++ ; } | *//локальные переменные потока 2*int i, j, k, **l=(NA/2)\*NB\*NC** ;for (i=**NA/2** ; i<**NA** ; i++) for (j=0 ; j<NB ; j++) for (k=0 ; k<NC ; k++) { D[l]= A[i]+B[j]+C[k] ; l++ ; } |

 Как и в примере 1, потоки не могут перезаписывать одни и те же переменные/ячейки массивов, значит добавление мьютексов не требуется.

 Часть комбинаторных задач в вариантах не может быть решена применением фиксированного количества циклов, в этой ситуации требуется реализация рекурсии.

 **Примечание**. Несколько вариантов включают применение *метода последовательных приближений*. Общий случай его реализации таков: задается точность e (например е=0.01), задача полностью вычисляется с некоторым шагом h1 и фиксируется результат R1, одновременно задача полностью вычисляется с некотором меньшим шагом h2 (например h2=h1/2.0) и фиксируется результат R2, если разница результатов не превышает точность е ( | R1-R2 | ≤ e ) то вычисления завершаются и любой из R1 и R2 объявляется искомым результатом, в противном случае выполняется новая итерация, на которой h1 устанавливается в h2 (h1=h2), h2 пересчитывается (h2=h1/2.0), вычисляются новые результаты R1 и R2, разность результатов сравнивается с точностью и т.д.

 Распараллеливание *метода последовательных приближений* может состоять в реализации двух потоков, где один вычисляет результат R1, второй R2, поток-родитель дожидается обеих, сравнивает результаты с точностью и по необходимости перезапускает оба потока с меньшим шагом. Можно заметить, что в этом случае второй поток получает задачу большей трудоемкости, и осуществить разделение на три потока, считающие: 1) R1 с шагом h на интервале [A, B], 2) R21 с шагом h/2.0 на части интервала [A, B], 2) R22 с шагом h/2.0 на оставшейся части интервала [A, B]. В случае значительного числа шагов вычислений в рамках одного интервала может быть целесообразным разделение на 4-16 потоков, часть которых обрабатывает участки интервала [A, B] в поисках R1, а другая часть - R2.