ВНИМАНИЕ!!!

Методические указания находятся в стадии создания, поэтому автор просит с пониманием отнестись к возможным опечаткам, неточностям, а также к неполноте изложенного материала.

Любые комментарии, исправления, дополнения и пожелания крайне желательны и будут приветствоваться.

Терехов Д.В.

Инструменты параллельного программирования

Методические указания к выполнению лабораторных работ

Белгород

2009

2 мар. 2010 г.

Содержание

Лабораторная работа № 1
Знакомство с технологией OpenMP

Цель

Ознакомиться с технологией OpenMP. Научиться компилировать OpenMP программы. Изучить директивы технологии OpenMP.

Теоретические сведения

OpenMP[1] — это набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с единой памятью на языках C, C++ и Fortran. OpenMP поддерживается многими коммерческими компиляторами на различных платформах.

В настоящее время опубликована спецификация OpenMP версии 3.0. Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и программного обеспечения, чья работа регулируется некоммерческой организацией, называемой OpenMP Architecture Review Board (ARB).

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы pragma и функции исполняющей среды OpenMP, которые подключаются как дополнительная библиотека. Директивы pragma, как правило, указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с #pragma omp. Как и любые другие директивы pragma, они игнорируются компилятором, не поддерживающим конкретную технологию — в данном случае OpenMP.

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API–функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции библиотеки OpenMP времени выполнения (исполняющей среды), в программу нужно включить заголовочный файл omp.h. Если

[1] от англ. Open Multi-Processing

вы используете в приложении только OpenMP–директивы pragma, включать этот файл не требуется.

Функции исполняющей среды OpenMP имеют префикс *omp_*. Директивы OpenMP имеют следующий формат:

```
#pragma omp <директива> [раздел [ [,] раздел]...]
```

Подробно со всеми директивами и их спецификациями можно ознакомиться в опубликованном стандарте OpenMP [1], сопроводительной документации к вашему компилятору, а также в учебном пособии Антонова А.С. «Параллельное программирование с использованием технологии OpenMP» [14].

Примеры программ

Рассмотрим пример параллельной программы, написанной с применением технологии OpenMP. Обратите внимание, в настройках некоторых компиляторов поддержка OpenMP по умолчанию может быть выключена. В этом случае при компиляции программы необходимо явно указать ключ использования OpenMP.

Пример – перемножение матриц. Распределим задачу по двум потокам следующим образом: разделим строки итоговой матрицы пополам, и каждый поток будет находить значения ячеек в тех строках, за которые он «отвечает».

```
#include <omp.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

/*
Процедура находит idxRow'ю строку произведения матриц
aMatrixA*aMatrixB согласованных размеров M*N*K
Результат помещается в соответствующую строку матрицы
aMatrixC
*/
void MatrixMul GetNthRow
     (
       double aMatrixA[], double aMatrixB[], double
aMatrixC[],
       unsigned M, unsigned N, unsigned K, unsigned idxRow
     )
{
  for(unsigned iCell= 0; iCell<K; iCell++)
  {
```

```
    double cell= 0;
    for(unsigned i= 0; i<N; i++)
      cell+= aMatrixA[idxRow*N + i] * aMatrixB[K*i + iCell];
    aMatrixC[idxRow*K + iCell]= cell;
  }
}

/*
Процедура перемножает матрицы aMatrixA*aMatrixB согласованных
размеров M*N*K
Результат помещается в матрицу aMatrixC
*/
void MatrixMul(double aMatrixA[], double aMatrixB[], double
aMatrixC[], unsigned M, unsigned N, unsigned K)
{
#pragma omp parallel for num_threads(2)
  /*
    Этой прагмой инициализируем параллельную часть цикла
    Каждая итерация цикла будет выполнена в одном из 2х
созданных потоков
    Распределение итераций по потокам будет выполнено
статически
  */
  for(int iRow=0; iRow<M; iRow++)
  {
    printf("%dth row is counting by %dth thread\n", iRow,
omp_get_thread_num());
    MatrixMul GetNthRow(aMatrixA, aMatrixB, aMatrixC, M, N,
K, iRow);
  }
}

/*
Инициализации матрицы случайными величинами
*/
void InitMatrix(double aMatrix[], unsigned ItemsCount)
{
  for(unsigned i= ItemsCount; i--; )
    aMatrix[i]= (int)(100 * rand()/(double)RAND_MAX);
}

void OutMatrix(double aMatrix[], unsigned M, unsigned N)
{
  for(unsigned i=0; i<M; i++)
  {
    for(unsigned j=0; j<N; j++)
      printf("%lf\t", aMatrix[i*N + j]);
    putchar('\n');
  }
  putchar('\n');
}
```

```
#define M 3
#define N 3
#define K 4

void main(void)
{
  double* MatrixA= new double [M*N];
  double* MatrixB= new double [N*K];
  double* MatrixC= new double [M*K];

  InitMatrix(MatrixA, M*N);
  InitMatrix(MatrixB, N*K);

  MatrixMul(MatrixA, MatrixB, MatrixC, M, N, K);
  OutMatrix(MatrixA, M, N);
  OutMatrix(MatrixB, N, K);
  OutMatrix(MatrixC, M, K);

  delete []MatrixA;
  delete []MatrixB;
  delete []MatrixC;

  getchar();
}

#undef M
#undef N
#undef K
```

Дополнительное чтение

Спецификации OpenMP (англ.) [1]

Статья в Википедии (русск.) [2]

Статья в Википедии (англ.) [3]

OpenMP и C++ (русск.) [4]

Начало работы с OpenMP (русск.) [5]

Параллельное программирование с использованием технологии OpenMP (русск.) [14]

Задание к выполнению лабораторной работы

Ознакомиться со средствами для организации параллельного выполнения программы, предоставляемыми технологией OpenMP. Изучить директивы синхронизации и балансировки нагрузки OpenMP.

Разработать параллельную программу, выполняющую решение задачи согласно варианту.

Выполнить замеры времени решения задачи параллельным и последовательным вариантами программы. Сравнить полученные результаты и объяснить их.

Вариант №1

Протабулировать функцию на заданном отрезке с заданным шагом.

Вариант №2

Выполнить сложение двух матриц одинакового размера.

Вариант №3

Найти сумму максимальных элементов строк матрицы.

Вариант №4

Найти площадь выпуклого многоугольника, заданного координатами вершин.

Вариант №5

Найти в данном тексте все палиндромы.

Вариант №6

Найти в тексте все вхождения данного образца.

Вариант №7

Дана последовательность вещественных чисел. Сократить количество десятичных разрядов после запятой каждого числа до двух.

Вариант №8

Дана последовательность арифметических выражений, операндами которых являются однозначные числа, а число операций не больше двух. Найти значения всех выражений.

Вариант №9

Дана матрица вещественных чисел. Преобразовать матрицу таким образом, чтобы элементы ее строк шли по убыванию.

Вариант №10

Вывести все согласные, которые отсутствуют в данном тексте.

Содержание отчета

1. Название и цель работы.
2. Задание к выполнению лабораторной работы согласно варианту.
3. Описание алгоритма решения задачи в виде блок–схем или словесное. Описание используемых параллельных методов вычислений.
4. Программа в виде исходных кодов (с поясняющими комментариями), а также в откомпилированном виде для демонстрации на ЭВМ.
5. Примеры работы программы на тестовых данных.
6. Выводы по работе.

Лабораторная работа № 2
Решение практических задач с применением технологии OpenMP

Цель

Изучить опции директив OpenMP. Получить практический навык использования технологии OpenMP при решении прикладных задач.

Теоретические сведения

Небольшой текст по обращению с локальными и глобальными переменными. Или вводный текст по опциям.

Подробно со всеми опциями и их спецификациями можно ознакомиться в опубликованном стандарте OpenMP [1], сопроводительной документации к вашему компилятору, а также в учебном пособии Антонова А.С. «Параллельное программирование с использованием технологии OpenMP» [14].

Примеры программ

Пример – перемножение матриц. Распределим задачу по двум потокам следующим образом: для нахождения элемента итоговой матрицы необходимо перемножить N пар чисел; каждый из 2х потоков будет делать половину этих перемножений, затем результат сложим и запишем в итоговую ячейку.

```
#include <omp.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

/*
Процедура перемножает матрицы aMatrixA*aMatrixB согласованных
размеров M*N*K
Результат помещается в матрицу aMatrixC
*/
void MatrixMul(double aMatrixA[], double aMatrixB[], double
aMatrixC[],
               unsigned M, unsigned N, unsigned K)
{
```

```
  double cell=0; // Объявляем переменную для сбора
перемноженных элементов

#pragma omp parallel num_threads(2)
  // Эта прагма инициализирует параллельные потоки
  // Конструкция, находящаяся под прагмой (в нашем случае
цикл
  // for(unsigned iRow=0; iRow<M; iRow++))
  // теперь выполнится в каждом из потоков
  for(unsigned iRow=0; iRow<M; iRow++)
  {
    for(unsigned iCell= 0; iCell<K; iCell++)
    {
      cell= 0;
#pragma omp sections reduction(+:cell)
// Эта прагма инициализирует параллельные секции.
// Теперь каждый из созданных ранее потоков выполнит только
одну из указанных
// секций. При этом, по завершении секций в переменную cell
будет занесена
// сумма локальных переменных cell каждого из потоков
      {
        { // Этот кусочек будет выполнен в первом потоке. Он
          printf("%dth thread begin to summaries firts
elements of %dth row\n", omp get thread num(), iRow);
          for(unsigned i= 0; i<N/2; i++)
            cell+= aMatrixA[iRow*N + i] * aMatrixB[K*i +
iCell];
        }
#pragma omp section
// Эта прагма инициализирует новую секцию, которая должна
выполнится в одном
// из потоков
        { // Этот кусочек будет выполнен во 2м потоке
          printf("%dth thread begin to summaries last
elements of %dth row\n", omp get thread num(), iRow);
          for(unsigned i= N/2; i<N; i++)
            cell+= aMatrixA[iRow*N + i] * aMatrixB[K*i +
iCell];
        }
      }
#pragma omp single
// Эта прагма инициализирует выполнение операции единственным
из потоков
// Запись в память будет осуществлена только одним из
потоков, не смотря на
// то, что команда находится в параллельной части программы
      aMatrixC[iRow*K + iCell]= cell;
    }
  }
}
```

```
/*
Инициализации матрицы случайными величинами
*/
void InitMatrix(double aMatrix[], unsigned ItemsCount)
{
  for(unsigned i= ItemsCount; i--; )
    aMatrix[i]= (int)(100 * rand()/(double)RAND MAX);
}

void OutMatrix(double aMatrix[], unsigned M, unsigned N)
{
  for(unsigned i=0; i<M; i++)
  {
    for(unsigned j=0; j<N; j++)
      printf("%lf\t", aMatrix[i*N + j]);
    putchar('\n');
  }
  putchar('\n');
}

#define M 3
#define N 3
#define K 4

void main(void)
{
  double* MatrixA= new double [M*N];
  double* MatrixB= new double [N*K];
  double* MatrixC= new double [M*K];

  InitMatrix(MatrixA, M*N);
  InitMatrix(MatrixB, N*K);

  MatrixMul(MatrixA, MatrixB, MatrixC, M, N, K);
  OutMatrix(MatrixA, M, N);
  OutMatrix(MatrixB, N, K);
  OutMatrix(MatrixC, M, K);

  delete []MatrixA;
  delete []MatrixB;
  delete []MatrixC;

  getchar();
}

#undef M
#undef N
#undef K
```

Приведенный пример демонстрирует использование опции reduction. В результате ее использования, переменной cell будет присвоена сумма локальных переменных cell каждого потока.

Задание к выполнению лабораторной работы

Ознакомиться со средствами OpenMP для работы с разделяемыми переменными и диспетчеризации потоков. Изучить опции директив технологии OpenMP.

Разработать параллельную программу (с использованием технологии OpenMP), решающую задачу из приведенного списка практических задач. Номер решаемой задачи выбрать соответственно своему варианту по следующему правилу: в качестве номера задачи принять номер варианта по модулю 10.

Выполнить замеры времени решения задачи параллельным и последовательным вариантами программы. Сравнить полученные результаты и объяснить их.

Содержание отчета

1. Название и цель работы.
2. Задание к выполнению лабораторной работы согласно варианту.
3. Краткие теоретические сведения по решаемой задаче.
4. Описание алгоритма решения задачи в виде блок–схем или словесное. Описание используемых параллельных методов вычислений.
5. Программа в виде исходных кодов (с поясняющими комментариями), а также в откомпилированном виде для демонстрации на ЭВМ.
6. Примеры работы программы на тестовых данных.
7. Выводы по работе.

Лабораторная работа № 3
Знакомство с технологией MPI

Цель

Ознакомиться с технологией MPI. Научиться компилировать и запускать MPI программы. Получить навык работы с простейшими средствами передачи сообщений между процессами.

Теоретические сведения

MPI[2] — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между экземплярами программы, выполняющими одну задачу (которые могут быть запущенными на различных компьютерах).

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI.

Практически все реализации MPI представляют собой внешнюю подключаемую библиотеку. В связи с этим, при компилировании MPI программ, компилятору необходимо дополнительно указывать заголовочные и библиотечные файлы.

Рассмотрим процесс создания параллельной программы средствами MPI. В качестве языка программирования возьмем C (C++), средой разработки послужит Visual Studio, пример будем собирать в ОС Microsoft Windows[3]. Создадим новый C++ проект, и добавим в него файл с исходным кодом MPI программы.

---

[2] от англ. Message Passing Interface — интерфейс передачи сообщений

[3] Стандарт MPI реализован практически на всех широко распространенных операционных системах. А так как реализация MPI представляет собой подключаемую библиотеку, то использовать MPI можно в любой современной среде разработки. В связи с этим, данный пример с незначительными изменениями повторяем в любой среде на любой платформе.

Так как MPI, как правило, распространяется отдельно от среды разработки, для компиляции проекта необходимо указать месторасположение заголовочных и lib файлов. К примеру, реализация MPI от Intel[4] по умолчанию распаковывает заголовочные файлы в папку *%Program files%\Intel\MPI\%ver%\%arch%\include*, а lib — *%Program files%\Intel\MPI\%ver%\%arch%\include*, где *%Program files%* — путь к папке с файлами программ на системном диске, *%ver%* — версия библиотеки MPI, *%arch%* — используемая для компиляции платформа.

В поставку MPI, как правило, включаются две версии lib файлов – отладочная и обыкновенная. В то время как обыкновенная служит для сборки финальных версий программ, оптимизированных на исполнение, отладочные версии позволяют собирать программы с дополнительной информацией, необходимой для отладки. В реализации MPI от Intel, к примеру, о том, что lib файл является отладочным, говорит присутствие буквы *d* в имени файла (impi.lib — обыкновенный файл, impid.lib — отладочный вариант).

Открываем свойства проекта и устанавливаем значения у следующих параметров:

Параметр	Значение	Примечание
Configuration Properties – C/C++ – General – Additional Include Directories	*%Program files%\Intel\MPI\%ver%\%arch%\include*	Путь к установленным заголовочным файлам
Configuration Properties – Linker – General – Additional Library Directories	*%Program files%\Intel\MPI\%ver%\%arch%\lib*	Путь к установленным lib файлам
Configuration Properties – Linker – General – Input	impicxxd.lib impid.lib	При сборке приложения использовать отладочные lib файлы.

[4] Ознакомительная версия Intel MPI распространяется бесплатно

После успешной сборки проекта получим исполняемый модуль (в примере ниже это Pi.exe). Следующий шаг – запустить этот исполняемый модуль на нескольких машинах одновременно. Для этих целей служит поставляемая вместе с реализацией MPI консольная утилита, которая обычно называется mpiexec.exe или mpirun.exe (для удобства использования с некоторыми реализациями может поставляться также оконная версия этой утилиты).

В качестве ключей командной строки при запуске этой утилиты указываются: имя запускаемой программы, имена станций, на которых необходимо запустить программу, а также прочие параметры. К примеру, для запуска 3–х экземпляров программы Pi.exe необходимо выполнить команду:

```
mpiexe.exe -n 3 pi.exe
```

После выполнения подобной команды, утилита mpiexec пробует связаться с указанными в командной строке станциями, на которых должен работать специальный демон[5] (как правило smpd), который уже запускает программу на станции, на которой работает сам. В связи с тем, что программа должна запускаться на различных станциях необходимо следить за тем, чтобы указанная в качестве ключа mpiexec программ была доступна демонам, запущенным на вычислительных станциях (это касается также входных данных, необходимых для работы запускаемой программы). Стандартным решением в этом случае является размещение программы и обрабатываемых данных в раскрытых папках, доступных всем вычислительным станциям

Примеры программ

В качестве примера, рассмотрим программу на C, вычисляющую значение числа Пи с некоторой наперед заданной точностью (данная программа входит в реализацию MPICH в качестве примера). Для вычисления приближенного значения числа π в данном примере используется соотношение:

[5] от *англ.* daemon - в системах класса UNIX — программа, работающая в фоновом режиме без прямого общения с пользователем. В терминологии Windows – это сервис.

$$\pi(n) \approx \frac{1}{n}\sum_{i=1}^{n}\frac{4}{1+\left(\frac{i-0.5}{n}\right)^2},$$

где n – некоторое положительно число (количество элементов ряда). Чем больше n, тем выше точность приближения.

Параллелизм программы основан на том, что i–й процесс из k запущенных вычисляет сумму каждого k–го слагаемого ряда, начиная с i–го (всего необходимо вычислить n слагаемых). После того как каждый процесс закончит вычисления, промежуточные результаты складываются.

```
// Подключаем необходимые заголовки
#include "mpi.h"
#include <stdio.h>
#include <math.h>

// Функция для промежуточных вычислений
double f(double a)
{
  return (4.0 / (1.0 + a*a));
}

// Главная функция программы
int main(int argc,char *argv[])
{
  // Объявляем переменные
  int done = 0, n, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264 3;
  double mypi, pi, h, sum, x;
  double startwtime = 0.0, endwtime;
  int namelen;
  char processor name[MPI MAX PROCESSOR NAME];

  // Инициализируем подсистему MPI
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI Comm rank(MPI COMM WORLD,&myid);
  MPI Get processor name(processor name,&namelen);

  /*
  // Выводим номер потока в общем пуле
  fprintf(stdout,"Process %d of %d is on %s\n",
  myid, numprocs, processor name);
  fflush(stdout);
  */

  while (!done) {
    // Если это головной процесс, спрашиваем пользователя о
```

```
    // количестве интервалов
    if (myid == 0) {
      fprintf(stdout, "Enter the number of intervals: (0
quits) ");
      fflush(stdout);
      if (scanf("%d",&n) != 1) {
        fprintf( stdout, "No number entered; quitting\n" );
        n = 0;
      }
      startwtime = MPI_Wtime();
    }
    // Рассылаем количество интервалов всем процессам
    //(в том числе и себе самому)
    MPI Bcast(&n, 1, MPI INT, 0, MPI COMM WORLD);
    if (n == 0)
      done = 1;
    else {
      h = 1.0 / (double) n;
      sum = 0.0;
      // Обсчитывает точки, "закрепленные" за процессом
      for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += f(x);
      }
      mypi = h * sum;

      // Собираем результаты со всех процессов и складываем
все
      MPI Reduce(&mypi, &pi, 1, MPI DOUBLE, MPI SUM, 0,
MPI COMM WORLD);

      // Если это головной процесс, выводим полученные
результаты
      if (myid == 0) {
        printf("pi is approximately %.16f, Error is %.16f\n",
pi, fabs(pi - PI25DT));
        endwtime = MPI Wtime();
        printf("wall clock time = %f\n", endwtime-
startwtime);
        fflush( stdout );
      }
    }
  }

  // Освобождаем подсистему MPI
  MPI Finalize();
  return 0;
}
```

Дополнительное чтение

Статья в Википедии (русск.) [6]

Статья в Википедии (англ.) [7]

Параллельное программирование с использованием технологии MPI (русск.) [8]

Страница MPI на parallel.ru (русск.)

Спецификации MPI (англ.) [9]

MPICH2 – бесплатная реализация MPI (англ.) — http://www.mcs.anl.gov/research/projects/mpich2/

Intell MPI – проприетарная реализация MPI (англ.) — http://software.intel.com/en-us/intel-mpi-library/

Задания к выполнению лабораторной работы

Ознакомиться с технологией MPI.

Откомпилировать поставляемую с реализацией MPI тестовую программу и запустить ее на различном числе процессоров. Написать к программе поясняющие комментарии.

Содержание отчета

1. Название и цель работы.
2. Задание к выполнению лабораторной работы.
3. Программа в виде исходных кодов (с поясняющими комментариями), а также в откомпилированном виде для демонстрации на ЭВМ.
4. Выводы по работе.

Лабораторная работа № 4
Средства MPI для обмена сообщениями

Цель

Ознакомиться со средствами технологии MPI для передачи сообщений между процессами и получить практический навык их использования.

Теоретические сведения

Основным средством коммуникации между процессами в MPI является передача сообщений друг другу. Собственно говоря, MPI — это и есть набор функций для передачи сообщений между процессами.

Все функции MPI имеют схожее название. Для C/C++, к примеру, все функции MPI начинаются с префикса *MPI_* (обратите внимание, что в связи со спецификой языка C, регистр в данном случае имеет значение).

Подробно со всеми функциями и их спецификациями можно ознакомиться в опубликованном стандарте MPI, сопроводительной документации к вашей реализации MPI, а также в учебном пособии Антонова А.С. «Параллельное программирование с использованием технологии MPI» [8].

Примеры программ

Рассмотрим пример простой программы на C++, осуществляющей обмен сообщениями между процессами. Обмен сообщениями сводится к тому, что главному процессу все остальные процессы отправляют следующую информацию: свой номер, общее количество процессов, имя станции, на которой запущен процесс.

Обратите внимание, так как программа написана на C++ (в отличие от предыдущего примера, написанного на C), используется объектная нотация и пространства имен.

```
#include "mpi.h"
#include <iostream>

int main (int argc, char *argv[])
{
  // Объявляем переменные
  int i, rank, size, namelen;
  char name[MPI MAX PROCESSOR NAME];
  MPI::Status stat;

  // Инициализируем подсистему MPI
  MPI::Init (argc, @argv);

  // Получаем информацию о процессе
  size = MPI::COMM_WORLD.Get_size ();
  rank = MPI::COMM_WORLD.Get_rank ();
  MPI::Get processor name (name, namelen);

  if (rank == 0) {
    // Находимся в головном процессе
    std::cout << "Hello world: rank " << rank << " of " <<
size << " running on " << name << "\n";

    // Собираем информацию от всех прочих процессов
    for (i = 1; i < size; i++) {
      MPI::COMM_WORLD.Recv (&rank, 1, MPI_INT, i, 1, stat);
      MPI::COMM_WORLD.Recv (&size, 1, MPI_INT, i, 1, stat);
      MPI::COMM WORLD.Recv (&namelen, 1, MPI INT, i, 1,
stat);
      MPI::COMM WORLD.Recv (name, namelen + 1, MPI CHAR, i,
1, stat);
      std::cout << "Hello world: rank " << rank << " of " <<
size << " running on " << name << "\n";
    }
  } else {
    // Находимся в дочернем процессе
    // Необходимо отправить информацию о себе головному
процессу
    MPI::COMM_WORLD.Send (&rank, 1, MPI_INT, 0, 1);
    MPI::COMM WORLD.Send (&size, 1, MPI INT, 0, 1);
    MPI::COMM WORLD.Send (&namelen, 1, MPI INT, 0, 1);
    MPI::COMM WORLD.Send (name, namelen + 1, MPI CHAR, 0, 1);

  }

  // Освобождаем подсистему MPI
  MPI::Finalize ();
  return (0);
}
```

Задания к выполнению лабораторной работы

Ознакомиться со средствами для обмена сообщениями между процессами, присутствующими в технологии MPI. Теоретический материал взять из [8], главы: «Основные понятия», «Общие процедуры MPI», «Передача/прием сообщений между отдельными процессами», «Коллективное взаимодействие процессов».

Разработать распределенную программу, выполняющую обмен сообщениями и провести исследование ее возможностей согласно своему варианту.

Вариант №1

Топология «звезда». Дочерние процессы пересылают пакеты данных заданного объема центральному процессу следующим образом: пакет рассылается каждым дочерним процессом двум соседям, откуда они пересылаются центральному процессу.

Оценить время рассогласованности получения пары пакетов центральным процессом от дочерних.

Вариант №2

Топология «звезда». Каждый дочерний процесс пересылает центральному произвольный пакет данных. Центральный процесс ретранслирует полученный от дочернего процесса пакет всем прочим процессам (в том числе и отправителю).

Оценить время пересылки единицы информации в зависимости от количества дочерних процессов.

Вариант №3

Топология «звезда». Центральный процесс рассылает поочередно дочерним процессам пакет данных произвольной длины. Дочерние процессы, получая от центрального пакет данных, отсылают его назад.

Оценить время пересылки единицы информации в зависимости от размера пакета.

Вариант №4

Топология «звезда». Центральный процесс пересылает всем дочерним процессам произвольный пакет данных, после чего центральным процессом назначается другой, и итерация повторяется.

Отранжировать процессы по скорости работы в качестве центрального процесса.

Вариант №5

Топология «кольцо». Процессы пересылают по кругу пакет данных, длина которого после каждой пересылки увеличивается до некоторого предела.

Оценить время пересылки единицы информации в зависимости от размера пакета.

Вариант №6

Топология «кольцо». Процессы пересылают по кругу в противоположных направлениях 2 пакета данных. Всякий раз, когда оба пакета оказываются в пространстве одного процесса, они увеличиваются в размере

Построить зависимость скорости роста пакетов от количества процессов.

Вариант №7

Топология «кольцо». Процессы пересылают по кругу пакет данных произвольной длины. После того как пакет сделает полный оборот, запускается другой пакет, но уже из другого процесса и итерация повторяется.

Оценить время оборота пакета в зависимости от количества процессов.

Вариант №8

Топология «два кольца». Все процессы условно делятся пополам. Каждая половина процессов пересылает по кругу пакет данных произвольной длины.

Оценить время оборота пакета в зависимости от количества процессов.

2 мар. 2010 г.

Вариант №9

Топология «конвейер». Головной процесс пересылает пакет данных произвольного размера через все процессы замыкающему. После каждой пересылки пакет увеличивается в размере.

Оценить время пересылки единицы информации в зависимости от количества процессов.

Вариант №10

Топология «конвейер». Головной процесс пересылает пакет данных произвольного размера через все процессы замыкающему. Замыкающий процесс, получив пакет данных, пересылает его тем же путем головному.

Оценить время пересылки единицы информации в зависимости от размера пакета.

Вариант №11

Топология «конвейер». Среди процессов выделяется один центральный процесс. Все остальные процессы условно делятся пополам и образуют два конвейера (слева и справа от центрального). Замыкающие процессы с каждой из сторон пересылают пакеты через процессы своего конвейера центральному.

Оценить скорость прохождения пакетов по левому и правому конвейеру в зависимости от размера пакета.

Вариант №12

Топология «две звезды». Среди процессов выделяется два ключевых, все остальные делятся поровну между выделенными в качестве подчиненных. Пакеты пересылаются по следующей схеме: дочерний процесс пересылает пакет своему центральному процессу, который в свою очередь пересылает его соседнему центральному процессу

Оценить скорость прохождения пакетов между центральными процессами в зависимости от количества процессов.

Вариант №13

Топология «две звезды». Среди процессов выделяется два ключевых, все остальные условно делятся поровну между

выделенными в качестве подчиненных. Каждый центральный процесс пересылает пакеты произвольной длины всем своим дочерним процессам.

Оценить время пересылки единицы информации в зависимости от размера пакета.

Вариант №14

Топология «песочные часы». Среди процессов выделяется центральный, а все остальные делятся пополам (будем считать, что одна половина из них располагается справа от центрального, а вторая слева). Каждый дочерний процесс пересылает пакеты данных избранному процессу с противоположной стороны, однако не напрямую, а через центральный процесс.

Оценить скорость передачи единицы информации в зависимости от количества процессов

Вариант №15

Топология «восьмерка». Среди процессов выделяется центральный, а все остальные делятся пополам (будем считать, что одна половина из них располагается справа от центрального, а вторая слева). Процессы с каждой стороны от центрального замыкаются с ним в кольцо. По каждому из полученных колец передается пакет.

Оценить скорость передачи единицы информации в зависимости от количества процессов

Содержание отчета

1. Название и цель работы.
2. Задание к выполнению лабораторной работы согласно варианту.
3. Топология обмена сообщениями между процессами в виде графа, а также словесное ее описание.
4. Программа в виде исходных кодов (с поясняющими комментариями), а также в откомпилированном виде для демонстрации на ЭВМ.
5. Примеры работы программы на тестовых данных.

2 мар. 2010 г.

6. Результаты исследования программы согласно варианту в виде таблиц или графиков с поясняющими комментариями.
7. Выводы по работе.

Лабораторная работа № 5
Решение практических задач с применением технологии MPI.

Цель

Получить практический навык использования технологии MPI при решении прикладных задач.

Примеры программ

Рассмотрим пример программы, осуществляющей перемножение матриц.

```
#include <mpi.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int c size, c rank;

double RunMatrixTest(unsigned mSize)
{
  srand(clock());
  // Выделяем память под матрицу, которую будем умножать на
саму себя
  double *matr= new double[mSize*mSize];
  // Выделяем память под матрицу-результат только в том
случае, если
  // находимся в головном процессе
  double *result= (!c rank)? new double[mSize*mSize]: NULL;
  // Выделяем память под одну строку матрицы результата
только если находимся в "рабочем" процессе
  double *row= (c_rank)? new double[mSize]: NULL;

  // Если находимся в головном процессе, инициализируем
исходную матрицу
  if(!c rank)
    for(unsigned i=mSize*mSize; i--; matr[i]=
rand()/(double)RAND_MAX);
  // Делимся матрицей со всем процессами. Если находимся в
головном
  // процессе, происходит рассылка, иначе - прием даннных
  MPI Bcast(matr, mSize*mSize, MPI DOUBLE, 0,
MPI COMM WORLD);
```

```
  double tm= MPI Wtime();

  // Вычисляем произведение матрицы на саму себя
  // Строки результирующей матрицы делятся между процессами:
0ю строку
  // считает первый,
  // 1ю - второй, 2ю- третий, 3ю - снова первый и т.д.

  // Матрицу обрабатываем блоками по c_size строк
  for(unsigned iBlockRow=0; iBlockRow<mSize; iBlockRow+=
c size)
  {
    // Номер строки в матрице, которую должен обработать
текущий
    // процесс для текущего блока
    unsigned iRow= iBlockRow+c_rank;
    if(iRow<mSize)
    {
      // Если находимся в головном процессе, то строка
приемник - это часть итоговой матрицы
      // Иначе, у нас под строку уже выделена память
      if(!c_rank)
        row= result+mSize*iRow;
      // Считаем результирующую строку
      for(unsigned iColumn=0; iColumn<mSize; iColumn++)
      {
        row[iColumn]= 0;
        for(unsigned k=0; k<mSize; k++)
          row[iColumn]+=
matr[iRow*mSize+k]*matr[k*mSize+iColumn];
      }

      MPI_Status status;
      if(c rank)
// Если находимся в "рабочем" процессе, отсылаем результат
головному
        MPI Send(row, mSize, MPI DOUBLE, 0, 0,
MPI_COMM_WORLD);
      else
// Если находимся в головном процессе, собираем результаты
расчетов "рабочих" процессов
        for(unsigned iProc=1; iProc<c size &&
(iBlockRow+iProc)<mSize; iProc++)
          MPI_Recv(result+(iBlockRow+iProc)*mSize, mSize,
MPI_DOUBLE, iProc, 0, MPI_COMM_WORLD, &status);
    }
    // Ждем, пока все процессы дойдут до этой строчки
    MPI Barrier(MPI COMM WORLD);
  }
  tm= MPI_Wtime()-tm;

  if(!c_rank)
```

```
  {
    //Если находимся в головном процессе, у нас имеется
полностью готовая матрица
  }

  delete[]matr;
  if(!c rank)delete[]result;
  if(c rank) delete[] row;

  return tm;
}

#define  SIZE 10
void main(int argc, char *argv[])
{
  // Инициализируем подсистему MPI
  MPI Init(&argc, &argv);

  // Получаем номер процесса и размер коммуникатора
  MPI Comm size(MPI COMM WORLD, &c size);
  MPI_Comm_rank(MPI_COMM_WORLD, &c_rank);

    try
  {
    // Запускаем процесс перемножения матриц
    double tm= RunMatrixTest(_SIZE);
    if(!c_rank)
      // Если находимся в головной процессе, выводим
результьтат
      printf("Multiplying matrixes (%dx%d): %lfsec",  SIZE,
 SIZE, tm);
  }
  __finally
  {
    // Освобождаем подсистему MPI
    MPI Finalize();
  }
}

#undef _SIZE
```

Задание к выполнению лабораторной работы

Разработать распределенную программу (с использованием технологии MPI), решающую задачу из приведенного списка практических задач. Номер решаемой задачи выбрать соответственно своему варианту по следующему правилу: в качестве номера задачи принять номер варианта +1 по модулю 10.

Содержание отчета

1. Название и цель работы.
2. Задание к выполнению лабораторной работы согласно варианту.
3. Краткие теоретические сведения по решаемой задаче.
4. Описание алгоритма решения задачи в виде блок–схем или словесное. Описание используемых параллельных методов вычислений.
5. Программа в виде исходных кодов (с поясняющими комментариями), а также в откомпилированном виде для демонстрации на ЭВМ.
6. Примеры работы программы на тестовых данных.
7. Выводы по работе.

Лабораторная работа № 6
Программирование графических процессоров (NVIDIA CUDA)

Цель

Ознакомиться с технологией NVIDIA CUDA. Научиться компилировать и запускать программы, содержащие CUDA код.

Теоретические сведения

CUDA[6] — технология GPGPU[7], позволяющая программистам выполнять вычислительные алгоритмы на графических процессорах ускорителей GeForce восьмого поколения и старше компании NVIDIA (которая и является разработчиком данной технологии).

Использующая технологию CUDA программа имеет два типа функций: функции host–кода — набор инструкций, который выполняет центральный процессор; и функций device–кода — набор инструкций, который выполняет процессором графического адаптера (как правило, файлы, содержащие исходный device–код, имеют расширение *.cu*). Фактически, технология позволяет программисту включать в программу специальные функции на C подобном языке, которые при запуске программы будут выполнены графическим процессором.

Технически эта возможность реализована с помощью специального компилятора *nvcc*, входящего в состав CUDA SDK[8]. Файлы device–кода (.cu) компилируются при помощи компилятора nvcc в объектные файлы, которые уже штатным сборщиком

[6] от англ. Compute Unified Device Architecture – унифицированная вычислительная архитектура (устройств)

[7] от англ. General-Purpose computing on Graphics Processing Units – использование графического процессора видеокарты для общих вычислений (вычисления, которые обычно выполняет центральный процессор)

[8] от англ. Software Development Kit - комплект средств разработки, в который, как правило, входят необходимые заголовочные файлы, компиляторы и различные утилиты.

собираются в исполняемый модуль наравне с остальным исходным кодом.

Выигрыш при использовании для общих вычислений графических процессоров достигается за счет того, что графический сопроцессор имеет в своем устройстве гораздо больше блоков ALU, нежели центральный процессор. В современном графическом процессоре счет блоков ALU[9] идет на сотни, в то время как в центральном процессоре редко когда бывает больше 4х вычислительных ядер.

С точки зрения программиста вычислительный блок CUDA представляет собой мультипроцессор из нескольких ядер, каждое из которых состоит из нескольких ALU. В итоге мультипроцессор целиком обладает сотнями ALU и тысячами регистров.

Подобная архитектура графического процессора, как в прочем и решаемые процессором задачи, не могут не накладывать ограничений на способ решения задач. Если вычислительные ядра центрального процессора полностью независимы – каждое из ядер выполняет свою последовательность команд, то все ALU вычислительного ядра графического мультипроцессора выполняют одну и ту же программу (различные ядра при этом могут выполнять различные последовательности инструкций, либо другую инструкцию той же самой последовательности). Каждое ALU вычислительного ядра в определенный момент времени выполняет ту же самую команду, что и все остальные ALU этого же ядра. Единственное, что отличается, — это операнды, над которыми выполняется эта команда.

Примеры программ

Тут пример программы перемножения матриц

Дополнительное чтение

Статья в Википедии (русск.) [10]

Домашняя страница CUDA (русск.) [11]

Документация по CUDA (англ.) [12]

[9] от англ. arithmetic and logic unit - блок процессора, который служит для выполнения логических и арифметических, преобразований над операндами

Задание к выполнению лабораторной работы

Ознакомится с технологией NVIDIA CUDA.

Откомпилировать поставляемую в составе CUDA SDK тестовую программу по перемножению матриц (matrixMul) и запустить ее.

Разработать программу на основе технологии CUDA, выполняющую табулирование функции (согласно своему варианту) на данном участке.

Вариант №1

$$f(x) = \sin(x) * \cos(x+2)^2,\ x = 0, 0.1, ..., 10$$

Вариант №2

$$f(x, y) = \sin(x) * x^y - 25xy,\ x = 0, 1, ..., 10,\ y = 0, 1, ..., 10$$

Вариант №3

$$f(x, y, z) = x^3 - y^2 + x^2 z^3 + z^2 \ln(x),\quad x = 0, 0.1, ..., 0.4,\quad y = 0, 0.1, ..., 0.4,\ y = 1, 2, ..., 10$$

Тут продолжение заданий

Содержание отчета

1. Название и цель работы.
2. Задание к выполнению лабораторной работы согласно варианту.
3. Описание алгоритма решения задачи в виде блок–схем или словесное. Описание используемых параллельных методов вычислений.
4. Программа в виде исходных кодов (с поясняющими комментариями), а также в откомпилированном виде для демонстрации на ЭВМ.
5. Примеры работы программы на тестовых данных.

2 мар. 2010 г.

6. Выводы по работе.

Расчетно–графическое задание
Решение задач с применением параллельных методов

Цель

Закрепить практический навык применения параллельных технологий при решении практических задач.

Задание к выполнению расчетно–графической работы

Решить задачу из приведенного перечня практических задач с использованием технологий параллельного программирования. При решении задачи выбрать оптимальные технологии параллельного программирования и обосновать свой выбор. Оптимизировать программу и добиться максимальной ее производительности.

Выполнить замеры времени решения задачи параллельным и последовательным вариантами программы. Сравнить полученные результаты и объяснить их

Номер решаемой задачи выбрать соответственно своему варианту по следующему правилу: в качестве номера задачи принять номер варианта + 2 по модулю 10.

Содержание отчета

1. Название и цель работы.
2. Задание к выполнению расчетно–графической работы согласно варианту.
3. Краткие теоретические сведения по решаемой задаче.
4. Описание используемых технологий параллельного программирования, а также обоснование использования этих технологий.

5. Описание алгоритма решения задачи в виде блок–схем или словесное. Описание используемых параллелизмов.
6. Программа в виде исходных кодов (с поясняющими комментариями), а также в откомпилированном виде для демонстрации на ЭВМ.
7. Примеры работы программы на тестовых данных.
8. Выводы по работе.

Практические задачи

Задача №1

Численно найти значение определенного интеграла данной функции с точностью до 10 знаков после запятой.

Теоретические сведения

Пусть необходимо найти значение интеграла $I = \int_a^b f(x)dx$.

Основная идея большинства методов численного интегрирования состоит в замене подынтегральной функции на более простую, интеграл от которой легко вычисляется аналитически. При этом для оценки значения интеграла получаются формулы вида: $\int_a^b f(x)dx \approx \sum_{i=1}^{n} \omega_i f(x_i)$,

где n — число точек, в которых вычисляется значение подынтегральной функции. Точки называются узлами метода, числа ω_i — весами узлов.

Одним из методов численного интегрирования является метод Симпсона, в котором подынтегральная функция на отрезке интегрирования заменяется параболой. Обычно в качестве узлов метода используют концы отрезка и его среднюю точку. В этом случае формула имеет очень простой вид: $\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$.

Если разбить интервал интегрирования на 2N равных частей, то получим:

$$\int_a^b f(x)dx \approx \frac{b-a}{6N}\left(f_0 + 4\left(f_1 + f_3 + \ldots + f_{2N-1}\right) + 2\left(f_2 + f_4 + \ldots + f_{2N-2}\right) + f_{2N}\right)$$

,

где $f_i = f\left(a + \frac{(b-a)i}{2N}\right)$ — значение функции в i–ой точке.

Приближение функции одним полиномом на всем отрезке интегрирования, как правило, приводит к большой ошибке в оценке значения интеграла. Для уменьшения погрешности отрезок интегрирования разбивают на части и применяют численный метод для оценки интеграла на каждом из них.

Величину погрешности приближения можно оценить, сравнив приближенные значения интеграла, полученные при разбиении отрезка интегрирования на различное число частей (как правило, шаг разбиения отрезков изменяется при этом вдвое).

Задача №2

Решить задачу Коши для системы обыкновенных дифференциальных уравнений первого порядка методом Рунге–Кутта.

Теоретические сведения

Пусть дана задача Коши для системы n дифференциальных уравнений первого порядка:

$$\begin{cases} y_1' = F_1(x, y_1, ..., y_n) \\ ... \\ y_n' = F_n(x, y_1, ..., y_n) \\ y_1(x^{(0)}) = y_1^{(0)} \\ ... \\ y_n(x^{(0)}) = y_n^{(0)} \end{cases},$$

где x — независимая переменная, $y_i' = F_i(x, y_1, ..., y_n)$ — производная i–ой неизвестной функции, $x^{(0)}$ — значение независимой переменной в начальный момент времени, $y_i(x^{(0)}) = y_i^{(0)}$ — значение i–ой неизвестной функции в начальный момент времени.

Зададимся некоторым ненулевым шагом интегрирования h. Тогда приближенное значение i–ой неизвестной функции в точке $x^{(k+1)} = x^{(k)} + h$ можно вычислить по формуле:

$$y_i(x^{(k+1)}) = y_i^{(k+1)} \approx y_i^{(k)} + \frac{h}{6}(k_1^{(i)} + 2k_2^{(i)} + 2k_3^{(i)} + k_4^{(i)}),$$

где

$$k_1^{(i)} = F_i(x^{(k)}, y_1^{(k)}, ..., y_n^{(k)})$$

$$k_2^{(i)} = F_i(x^{(k)} + h/2, y_1^{(k)} + k_1^{(1)}/2, ..., y_n^{(k)} + k_1^{(n)}/2)$$

$$k_3^{(i)} = F_i(x^{(k)} + h/2, y_1^{(k)} + k_2^{(i)}/2, ..., y_n^{(k)} + k_2^{(n)}/2)$$

$$k_4^{(i)} = F_i(x^{(k)} + h, y_1^{(k)} + k_3^{(i)}, ..., y_n^{(k)} + k_3^{(n)})$$

Задача №3

Выполнить локальную фильтрацию растрового изображения.

Теоретические сведения

Для представления графической информации на двумерной плоскости монитора применяется растровая графика. Растровая графика оперирует с произвольными изображениями в виде растров. Растр[10] — это описание изображения на плоскости путем разбиения (дискретизации) его на одинаковые элементы по регулярной сетке и присвоением каждому элементу цветовой информации.

Наиболее распространенная цветовая модель — RGB[11]. В этой модели каждый пиксель представлен тремя числами — тремя цветовыми составляющими (красной, зеленой и голубой).

Локальная фильтрация — это метод обработки растровых изображений, позволяющий улучшить его качества: уменьшить искажения, увеличить резкость, выделить контуры и т.д. Суть метода заключается в перемещении по изображению локального окна (матрицы). Окно проходит по изображению так, что каждый пиксель

[10] от англ. Raster

[11] От англ. red, green, blue — красный, зеленый, голубой

один раз бывает его центром. На окне определена весовая функция $H(p,q)$, где p,q — координаты центрального пикселя окна. Эта весовая функция определяет новый цвет пикселя, который есть центр окна. Эффект фильтрации зависит только от функции H. Размер окна обычно бывает 3x3, 5x5 или 7x7 и т.д.

Например, окно для сглаживания изображения, имеет следующий вид:

$$A[i,j]=\frac{1}{8}\begin{vmatrix}0 & 1 & 0\\ 1 & 4 & 1\\ 0 & 1 & 0\end{vmatrix}$$

$D=\frac{1}{8}$ — коэффициент

$C'[x,y]=D\sum_{i,j}\left(A[i,j]*C[i,j]\right)$ — правило получения нового значения цвета пикселя

$C_{n\times m}$ — матричное представление растра.

Задача №4.

Найти произведение N матриц согласованных размеров.

Теоретические сведения

Пусть даны две матрицы $A_{n\times m}$ и $B_{m\times k}$:

$$A_{n\times m}=\begin{pmatrix}a_{11} & a_{12} & \dots & a_{1m}\\ a_{21} & a_{22} & \dots & a_{2m}\\ & & \dots & \\ a_{n1} & a_{n2} & \dots & a_{nm}\end{pmatrix},\ A_{m\times k}=\begin{pmatrix}a_{11} & a_{12} & \dots & a_{1k}\\ a_{21} & a_{22} & \dots & a_{2k}\\ & & \dots & \\ a_{m1} & a_{m2} & \dots & a_{mk}\end{pmatrix}.$$

Произведение матриц $A_{n\times m}$ и $B_{m\times k}$ называется такая матрица $C_{n\times k} = A_{n\times m} \times B_{m\times k}$, каждый элемент которой равен: $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$.

Операция перемножения матриц определена только для матриц согласованного размера (число столбцов левой матрицы должно совпадать с числом строк в правой) и не обладает свойством коммутативности: $A_{n\times m} \times B_{m\times k} \neq B_{m\times k} \times A_{n\times m}$.

Задача №5

Решить систему линейных алгебраических уравнений методом Гаусса–Жордана.

Теоретические сведения

Пусть дана система из n линейных алгебраических уравнений от n неизвестных:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}.$$

Матрицу коэффициентов данной системы можно записать в виде:

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ & \ldots & & & \ldots \\ a_{n1} & a_{n2} & \ldots & a_{nn} & b_n \end{array}\right),$$

где последний столбец – свободные коэффициенты.

Суть метода заключается в приведении матрицы коэффициентов с учетом свободных членов сначала к верхнему треугольному, а затем к единичному виду:

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ & \dots & & & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 1 & a_{12}' & \dots & a_{1n}' & b_1' \\ 0 & 1 & \dots & a_{2n}' & b_2' \\ & & \dots & & \dots \\ 0 & 0 & \dots & 1 & b_n' \end{array}\right) \rightarrow \left(\begin{array}{cccc|c} 1 & 0 & \dots & 0 & b_1'' \\ 0 & 1 & \dots & 0 & b_2'' \\ & & \dots & & \dots \\ 0 & 0 & \dots & 1 & b_n'' \end{array}\right).$$

В последнем столбце после всех преобразований будут искомые неизвестные.

Алгоритм метода:

1. Выбирается первая колонка слева, в которой есть хоть одно отличное от нуля значение.
2. Если самое верхнее число в этой колонке есть нуль, то меняется вся первая строка матрицы с другой строкой матрицы, где в этой колонке нет нуля.
3. Все элементы первой строки делятся на верхний элемент выбранной колонки.
4. Из оставшихся строк вычитается первая строка, умноженная на первый элемент соответствующей строки, с целью получить первым элементом каждой строки (кроме первой) нуль.
5. Далее проводим такую же процедуру с матрицей, получающейся из исходной матрицы после вычёркивания первой строки и первого столбца.
6. После повторения этой процедуры (n-1) раз получаем верхнюю треугольную матрицу
7. Вычитаем из предпоследней строки последнюю строку, умноженную на соответствующий коэффициент, с тем, чтобы в предпоследней строке осталась только 1 на главной диагонали.
8. Повторяем предыдущий шаг для последующих строк. В итоге получаем единичную матрицу и решение на месте свободного вектора (с ним необходимо проводить все те же преобразования).
9. Чтобы получить обратную матрицу, нужно применить все операции в том же порядке к единичной матрице.

Задача №6

Найти матрицу, обратную данной квадратной, с помощью союзной матрицы.

Теоретические сведения

Пусть дана квадратная матрица: $A_{n\times n} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & & \dots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$.

Обратной называется такая матрица $A_{n\times n}{}^{-1}$, для которой выполнено условие: $A_{n\times n} \times A_{n\times n}{}^{-1} = E$, где E — единичная матрица.

Для поиска обратной матрицы можно воспользоваться равенством:

$$A_{n\times n}{}^{-1} = \frac{1}{\det A} \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ & & \dots & \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix},$$

где $\det A$ — определитель матрицы A, $A_{ij} = (-1)^{i+j} M_j^i$ — алгебраическое дополнение элемента a_{ij} матрицы A, M_j^i — минор (определитель матрицы, получающийся из A вычеркиванием i–й строки и j–го столбца).

Задача №7

Реализовать алгоритм блочного шифрования в режиме электронной кодовой книги (простой замены).

Теоретические сведения

Алгоритм блочного шифрование перерабатывает открытый текст в шифротекст блоками фиксированного размера. Размер блока зависит от самого алгоритма или его настроек.

Открытый текст некоторой длинны с помощью блочного алгоритма шифрования можно переработать в шифротекст различными способами (режимами). Самым простым из них (и самым

небезопасным) является режим электронной кодовой книги. При работе в этом режиме, каждый блок шифротекста получается путем шифрования соответствующего блока открытого текста по данному ключу.

Задача №8

Найти матрицу, обратную данной квадратной методом Гаусса.

Пусть дана квадратная матрица: $A_{n\times n} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & & \dots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$.

Обратной называется такая матрица $A_{n\times n}^{-1}$, для которой выполнено условие: $A_{n\times n} \times A_{n\times n}^{-1} = E$, где E — единичная матрица.

Возьмем две матрицы: саму матрицу $A_{n\times n}$ и единичную $E_{n\times n}$. Приведём матрицу $A_{n\times n}$ путем элементарных преобразований к единичной матрице методом Гаусса (см. задачу №5). После применения каждой операции к первой матрице применим ту же операцию ко второй матрице. Когда приведение первой матрицы к единичному виду будет завершено, вторая матрица окажется равной искомой обратной матрице $A_{n\times n}^{-1}$.

Задача №9

Найти кратчайшие пути между всеми вершинами графа с помощью алгоритма Флойда.

Алгоритм Флойда–Уоршелла (рус.) — http://ru.wikipedia.org/wiki/Алгоритм_Варшалла

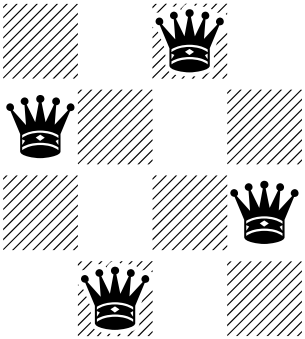
Задача №10

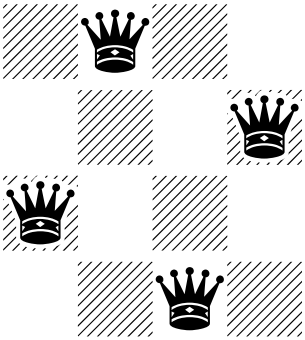
Найти все возможные размещения N ферзей на шахматной доске размеров NxN.

2 мар. 2010 г.

Теоретические сведения

Рассмотрим шахматную доску, которая имеет размеры не 8x8, а NxN, где N>0. Как известно, шахматный ферзь атакует все клетки и фигуры на одной с ним вертикали, горизонтали и диагонали. Любое расположение нескольких ферзей на шахматной доске будем называть их размещением. Размещение называется допустимым, если ферзи не атакуют друг друга. Размещение N ферзей на шахматной доске NxN называется полным. Допустимые полные размещения существуют не при любом значении N. Например, при N=2 или 3 их нет. При N=4 их лишь 2, причем они зеркально отражают друг друга.





Библиографический список

1. OpenMP API specification [Электронный ресурс, англ.]: спецификации OpenMP – режим доступа http://openmp.org/wp/openmp-specifications/.

2. OpenMP [Электронный ресурс]: Материал из Википедии — свободной энциклопедии / Авторы Википедии // Википедия, свободная энциклопедия. – Сан–Франциско: Фон Викимедия, 2009.— Режим доступа: http://ru.wikipedia.org/wiki/OpenMP, — свободный.

3. OpenMP [Электронный ресурс, англ.]: Материал из Википедии — свободной энциклопедии / Авторы Википедии // Википедия, свободная энциклопедия. — Сан–Франциско: Фон Викимедия, 2009.— Режим доступа: http://en.wikipedia.org/wiki/OpenMP, — свободный.

4. OpenMP и C++ [Электронный ресурс, англ.] / Канг Су Гэтлин, Пит Айсенси // MSDN Magazine. — 2005. — Октябрь. — Режим доступа: http://www.microsoft.com/Rus/Msdn/Magazine/2005/10/OpenMP.mspx

5. Начало работы с OpenMP [Электронный ресурс] / Richard Gerber // Intel. — Режим доступа: http://software.intel.com/ru-ru/articles/getting-started-with-openmp/

6. Message Passing Interface (MPI) [Электронный ресурс]: Материал из Википедии — свободной энциклопедии / Авторы Википедии // Википедия, свободная энциклопедия. — Сан–Франциско: Фон Викимедия, 2009.— Режим доступа: http://ru.wikipedia.org/wiki/MPI, — свободный.

7. Message Passing Interface (MPI) [Электронный ресурс, англ.]: Материал из Википедии — свободной энциклопедии / Авторы Википедии // Википедия, свободная энциклопедия. — Сан–Франциско: Фон Викимедия, 2009.— Режим доступа: http://en.wikipedia.org/wiki/Message_Passing_Interface, — свободный.

8. *Антонов, А.С.* Параллельное программирование с использованием технологии MPI / А.С. Антонов // Издательство московского университета, 2004.— ISBN 5-211-04907-1

9. MPI Documents [Электронный ресурс, англ.]: спецификации MPI — режим доступа: http://www.mpi-forum.org/docs/docs.html

10. CUDA [Электронный ресурс]: Материал из Википедии — свободной энциклопедии / Авторы Википедии // Википедия, свободная энциклопедия. — Сан–Франциско: Фон Викимедия, 2009.— Режим доступа: http://ru.wikipedia.org/wiki/CUDA, — свободный.

11. Домашняя страница CUDA [Электронный ресурс] // NVIDIA.— Режим доступа: http://www.NVIDIA.ru/object/cuda_learn_ru.html

12. Документация по CUDA [Электронный ресурс, англ.] // NVIDIA.— Режим доступа: http://www.NVIDIA.ru/object/cuda_develop_emeai.html

13. *Берилло А.* NVIDIA CUDA — неграфические вычисления на графических процессорах / А. Берилло // ixbt.com, 23.09.2008. Режим доступа: http://www.ixbt.com/video3/cuda-1.shtml, http://www.ixbt.com/video3/cuda-2.shtml

14. *Антонов, А.С.* Параллельное программирование с использованием технологии OpenMP / А.С. Антонов // Издательство московского университета, 2009. — ISBN 978-5-211-05702-9