# Введение в параллельные вычисления

## Лекция 10. Алгоритмы распараллеливания и отладка параллельных программ

КС-40, КС-44
РХТУ

Преподаватель
Митричев Иван Игоревич, к.т.н.,
ассистент кафедры ИКТ

2017

# Виды параллелизма

**Параллелизм данных (data parallelism)** – каждый поток/процесс обрабатывает свою часть данных.
Пример: сложение двух массивов размерностью MxM. Каждый поток/процесс складывает N строк первого и N строк второго массива. N<M.

**Параллелизм задач (task parallelism)** – каждый поток/процесс выполняет независимую задачу.
Пример: вычисление оценок коэффициентов линейной регрессионной модели

$$\hat{b}_{OLS} = (X^T X)^{-1} X^T y$$

Первый поток/процесс находит произведение матриц $X^TX$, второй поток/процесс находит произведение матриц $X^TY$.

# Языки и надстройки над языками для параллельных вычислений

**DVM – проект ИПМ им. Келдыша РАН, расширяет языки C и Fortran.**

Пример использования метода Якоби для решения уравнения Лапласа

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define Max(a,b) ((a)>(b)?(a): (b))
#define DVM(dvmdir)
#define DO(v,l,h,s)   for(v=l; v<=h; v+=s)
#define  L  8
#define  ITMAX 20
int i,j,it,k;
double eps;
double MAXEPS = 0.5;
FILE *f;
/* 2D arrays block distributed along 2 dimensions */
DVM(DISTRIBUTE [BLOCK][BLOCK]) double A[L][L];
DVM(ALIGN[i][j] WITH A[i][j]) double B[L][L];
int main(int argn, char **args)
{
/* 2D loop with base array A */
DVM(PARALLEL [i][j] ON A[i][j])
DO(i,0,L-1,1)
DO(j,0,L-1,1)
     {A[i][j]=0.;
     B[i][j]=1.+i+j;
     }

/******  iteration loop **************************/
DO(it,1,ITMAX,1)
{
  eps= 0.;
/* Parallel loop with base array A */
/* calculating maximum in variable eps */
DVM(PARALLEL [i][j] ON A[i][j]; REDUCTION MAX(eps))
DO(i,1,L-2,1)
DO(j,1,L-2,1)
          {eps =  Max(fabs(B[i][j]-A[i][j]),eps);
          A[i][j] = B[i][j];
          }
/* Parallel loop with base array  B  and */
/* with prior updating shadow elements of array A  */
DVM(PARALLEL[i][j] ON B[i][j]; SHADOW_RENEW A)
DO(i,1,L-2,1)
DO(j,1,L-2,1)
   B[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4.;

  printf("it=%4i   eps=%3.3E\n", it,eps);
  if (eps < MAXEPS) break;
  }/*DO it*/
f=fopen("jacobi.dat","wb");
fwrite(B,sizeof(double),L*L,f);
return 0;
}
```

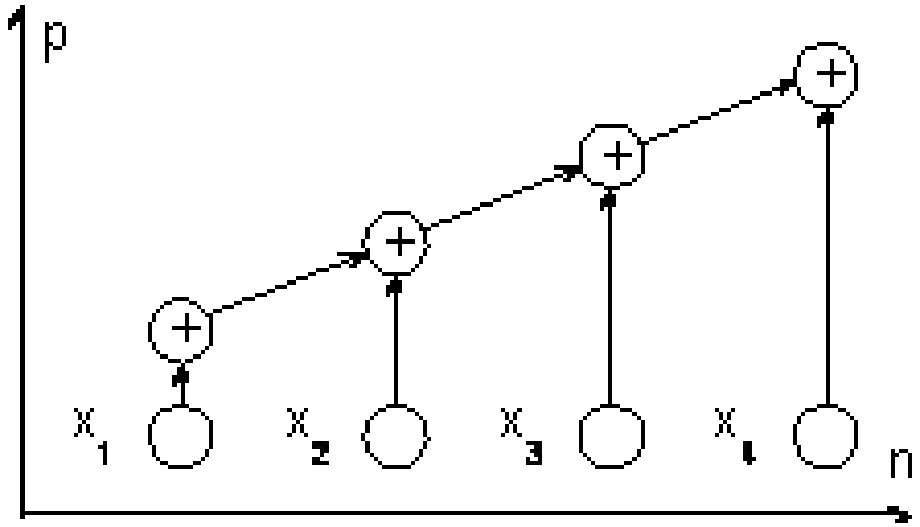
***Пример с сайта http://www.keldysh.ru/dvm/dvmhtm1107/rus/usr/cdvm/cdvmLDr3.html***

# Языки и надстройки над языками для параллельных вычислений

Erlang – язык программирования с поддержкой параллельных вычислений на основе модели акторов.

Модель акторов: актор – сущность, которая может принять сообщение от других акторов и, как результат,
1) отправить конечное число сообщений другим акторам; 2) выбрать режим обработки следующих сообщений; 3) создать конечное число новых акторов.

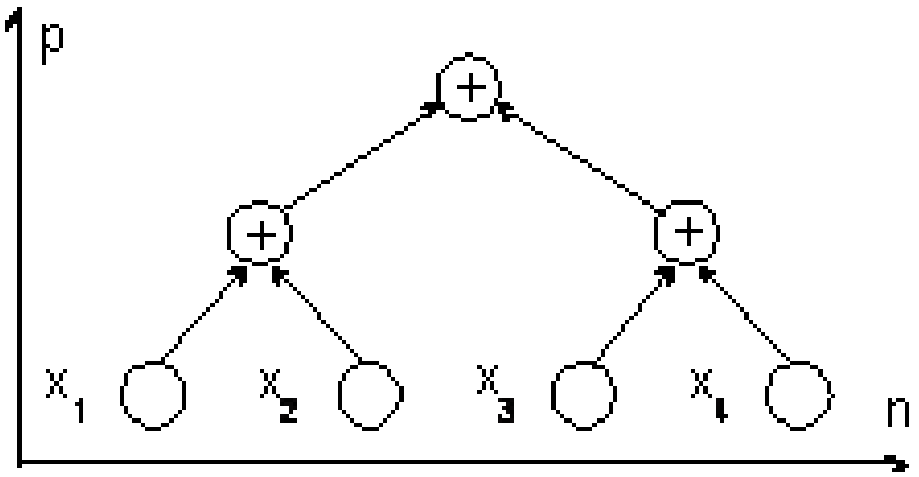
# Последовательная схема суммирования




Количество итераций алгоритма, необходимое для выполнения

$$k = n-1$$

# Каскадная схема суммирования




Количество итераций алгоритма (время работы пропорционально ему)

$$k = \log_2 n$$

Показатель ускорения

$$S_p = T_1 / T_p = (n-1) / \log_2 n,$$

Показатель эффективности (учитывает количество процессоров!)

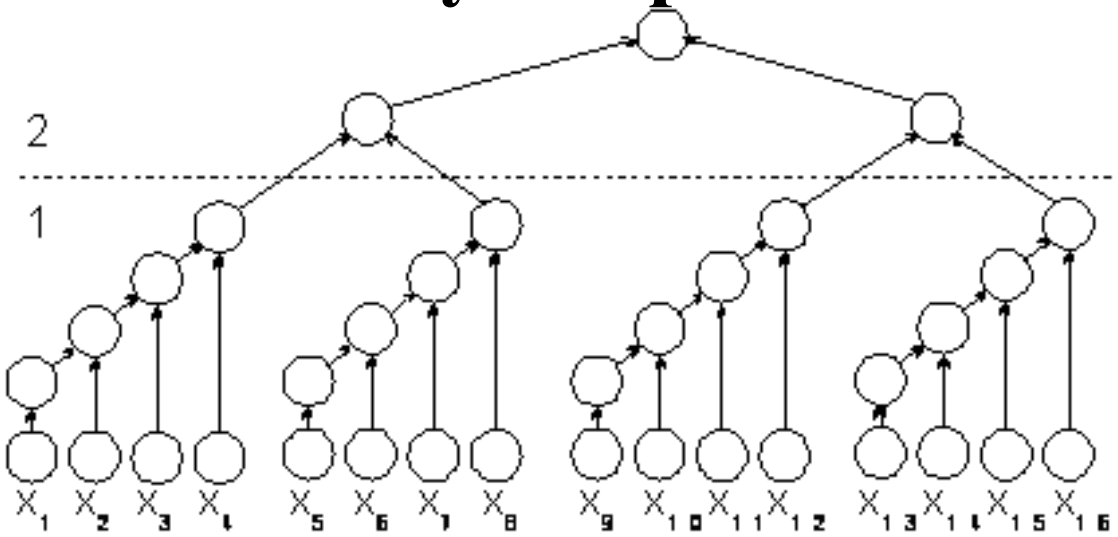
$$E_p = T_1 / pT_p = (n-1) / (p \log_2 n) = (n-1) / ((n/2) \log_2 n),$$

Общее количество операций суммирования вновь n-1, как в последовательном.

Всего требуется процессоров p = n/2

Недостаток: $\lim E_p \to 0 \quad при \quad n \to \infty$

# Модифицированная каскадная схема суммирования




Разделим n элементов на $(n/\log_2 n)$ групп по $\log_2 n$ элементов. Применим последовательный алгоритм. Требуется около $\log_2 n$ операций, а процессоров – по числу групп. Второй этап – суммирование результатов, полученных в группах. Всего Операций $\log_2(n/\log_2 n) \le \log_2 n$ на $p_2 = (n/\log_2 n)/2$ (число процессоров для каскадной схемы)

Показатель ускорения

$$S_p = T_1 / T_p = (n-1)/2\log_2 n,$$

Показатель эффективности

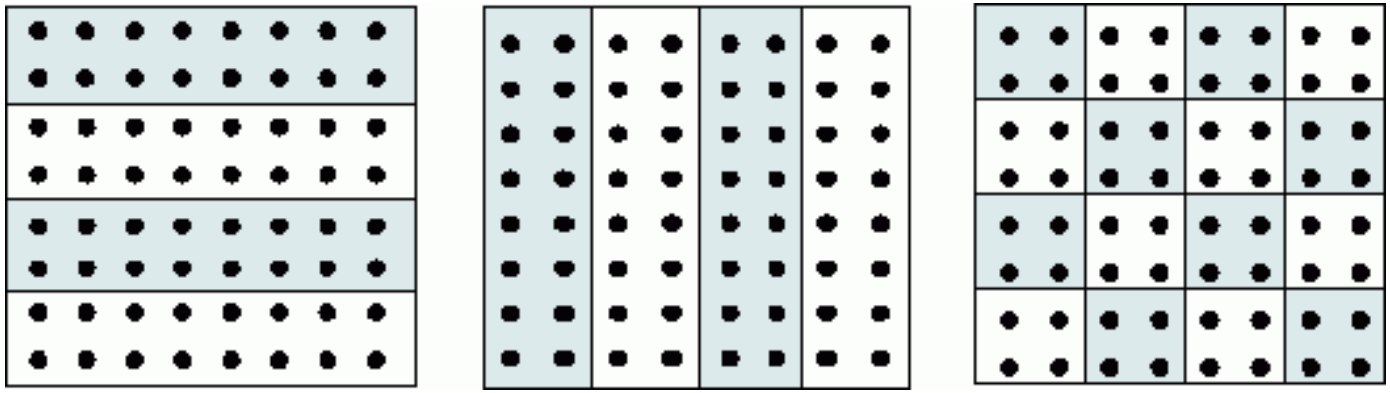
$$E_p = T_1 / pT_p = (n-1)/(2(n/\log_2 n)\log_2 n) = (n-1)/2n.$$

Требуемое время больше, чем в каскадном $T_p = 2\log_2 n$

Преимущество: $\lim E_p \to 0.5 \quad при \quad n \to \infty$

# Способы разделения данных при хранении матрицы

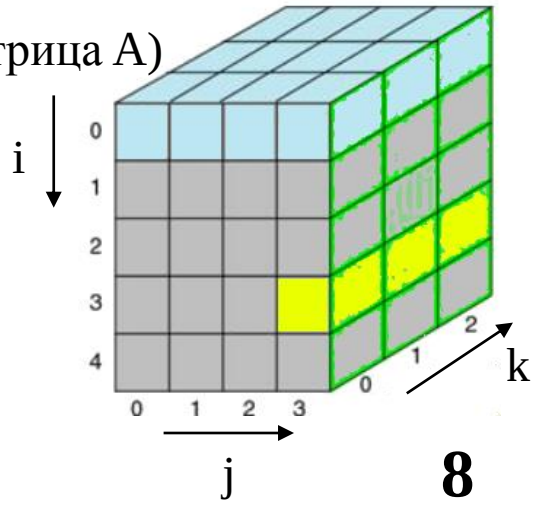
Ленточная (а – по строкам, б – по столбцам) и блочная (в) схемы



а б в

Для упрощения кода удобно хранить многомерные данные (матрица A) в виде одномерных массивов (W). Для трехмерного случая

$$A_{i,j,k} = W_{i*Nj*Nk + j*Nk + k}$$




Nj*Nk – количество элементов в слое (слой выделен голубым)
Nk – количество элементов в ряду (ряд выделен желтым)

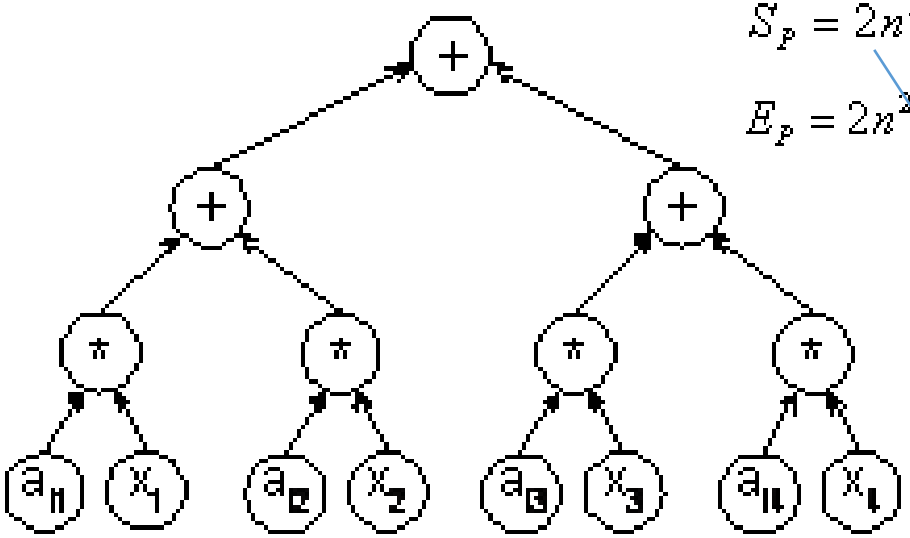
При этом применяется ленточная форма разделения данных по вычислительным процессорам.

# Умножение матрицы на вектор

Пусть i-ая группа процессоров умножает элементы i-ой строки матрицы A на элементы вектора x (ОДНО действие). Затем, по каскадной схеме суммирует результаты:

$p = n^2$

$T_p = 1 + \log_2 n$

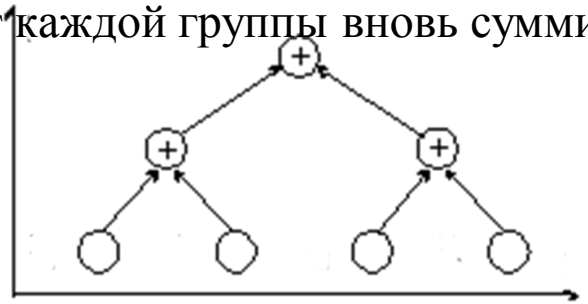


$S_p = 2n^2 / (1 + \log_2 n)$

$E_p = 2n^2 / p(1 + \log_2 n) = 2/(1 + \log_2 n)$

$2n^2$ сложений и умножений в последовательном алгоритме

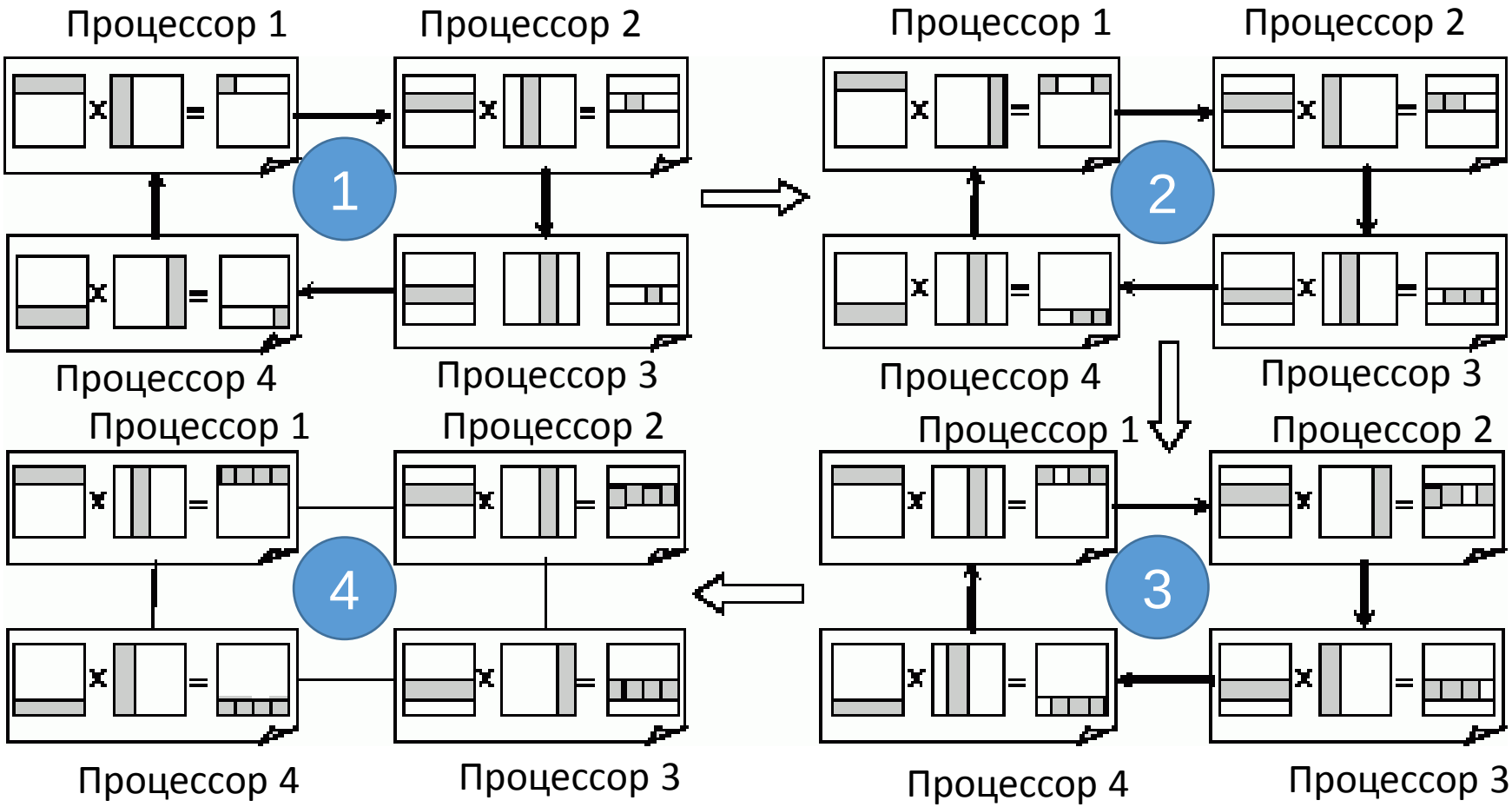
После чего результаты от каждой группы вновь суммируются по каскадной схеме.



# Перемножение матриц AxB

1) Каждый процессор получает свою строку матрицы A (при умножении слева)
2) На каждой итерации каждому процессору рассылается ровно один столбец матрицы B под номером $M_j$. всем разные. На следующей итерации процессор получает $M_j$+1 столбец (то есть следующий вправо в матрице B), или первый, если $M_j$+1-го столбца в матрице B нет.

***При этом хранение матриц производят в ленточной форме.***



# Схема «Мастер» - «рабочие»

Один процесс/поток является мастером. Который рассылает работы всем остальным и следит за ее выполнением. Возможны варианты:

1) Мастер нагружает поровну работой всех рабочих (статическая балансировка)
2) Мастер нагружает новой работой тех рабочих, которые закончили предыдущую работу (динамическая балансировка)

```
#include <stdio.h>
#include "mpi.h"
#define N 1000
#define MAXPROC 128
void slave(double *a, int n)
{
/* обработка локальной части массива a */
}
void master(double *a, int n)
{
/* обработка массива a */
}
int main(int argc, char **argv)
{
  int rank, size, num, k, i, indices[MAXPROC], source;
  MPI_Request req[MAXPROC];
  MPI_Status statuses[MAXPROC];
  double a[N][MAXPROC];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank!=0)
    while(1){
      slave((double*)a, N);
      MPI_Send(a, N, MPI_DOUBLE, 0, 5,
MPI_COMM_WORLD);
    }
    else{
      for(i = 0; i<size-1; i++)
        MPI_Irecv(&a[0][i], N, MPI_DOUBLE, i, 5,
MPI_COMM_WORLD, &req[i]);
      while(1){
        MPI_Waitsome(size-1, req, &num, indices, statuses);
        for (i = 0; i< num; i++){
          source = statuses[i].MPI_SOURCE;
          master(&a[0][source], N);
          MPI_Irecv(&a[0][source], N, MPI_DOUBLE, source, 5,
MPI_COMM_WORLD, &req[source]);
        }
      }
    }
  MPI_Finalize();  }
```

http://parallel.ru/sites/default/files/tech/tech_dev/MPI%26OpenMP/example12a.c

# Схема «Мастер» - «рабочие» с мастером-рабочим

```
#include <iostream>
#include "mpi.h"
#define N 64
#define MAXPROC 4
void slave(double *a, int n, int rank)
{
        for (int i=0; i<n; i++)
                a[i]=i+rank;
}

void master(double* a, int num, int portion)
{
        for (int i=0; i<portion; i++)

        std::cout<<a[num*portion+i]<<std::endl;
}

int main(int argc, char **argv)
{
  int rank, size, source, d;
  MPI_Request req[MAXPROC-1];
  MPI_Status statuses[MAXPROC-1];
  double a[N];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int slaves= size - 1;
  double* b = new double [N/size];

  MPI_Scatter(a, N/size, MPI_DOUBLE, b,
        N/size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if(rank!=0)
    {
      slave((double*)b, N/size, rank);
                d=1; //send some flag
                MPI_Send(&d, 1, MPI_INT, 0, 5,
MPI_COMM_WORLD);
              MPI_Gather(b, N/size, MPI_DOUBLE, a, N/size,
MPI_DOUBLE, 0,
       MPI_COMM_WORLD);
    }
    else{
      for(int i = 1; i<size; i++)
        MPI_Irecv(&d, 1, MPI_INT, i, 5, MPI_COMM_WORLD,
&req[i-1]);

     MPI_Waitall(size-1, req, statuses);
// to make Gather and Scatter work, master should work as slave
              slave((double*)b, N/size, rank);
              MPI_Gather(b, N/size, MPI_DOUBLE, a, N/size,
MPI_DOUBLE, 0,
       MPI_COMM_WORLD);

        for (int i = 0; i< size-1; i++){
          source = statuses[i].MPI_SOURCE;
          master(a,source,N/size);
                master(a,0,N/size); //print own part of array
        }
    }
  delete [] b;
  MPI_Finalize();
}
```

**l10_01.cpp**

# Отладка параллельных программ

**Отладчик gdb.**

gdb ./a.out – загрузка программы в отладчик. Или запустить gdb, в нем: file ./a.out
r или run – запуск программы.
b или breakpoint – точка останова
b файл:строка – установить точку останова на заданную строку кода
s или step – шаг с заходом внутрь функций, n или next – шаг без захода
d или delete удалит точки (можно указать конкретную точку как параметр)
list файл/имя_функции/номер-строки – показать код. Можно указывать через запятую диапазон строк.
break строка if x > 5 – остановить, если выполнено x>5
break строка thread номер потока – остановиться на строке в потоке (info threads – узнать номера потоков)
watch x > 5 – остановиться, когда условие x>5 станет истинным
u или up, d или down – перемещение по стеку вызовов функций
thread 10 – переключиться в поток 10
Ctrl-c – прервать исполнение
c или continue – продолжить исполнение
p переменная – вывести значение переменной (p или )
q или quit – выйти (y для подтверждение выхода)
sudo gdb attach номер_процесса – присоединить отладчик к уже работающему процессу в ОС (в Linux: ps aux – список процессов)

См. http://www.yolinux.com/TUTORIALS/GDB-Commands.html#GDB_COMMANDS

# Отладка параллельных программ

**Для включения исходного кода в программу и облегчения отладки, обязательно при компиляции включите опцию -g** (gcc -g, g++ -g, mpicxx -g, mpic++ -g, mpicc -g)

```
mpicxx -g l10_02.cpp
gdb ./a.out
b l10_02.cpp:10
r
```

```
g++ -g -O0 -fopenmp -lgomp l10_03.cpp
gdb ./a.out
b l10_03.cpp:11
r
info threads
```

**l10_02.cpp =** l08_01

**l10_03.cpp =** l04_08

**l10_04.cpp =** lec02_11

```
g++ -g -O0 -std=c++11 -pthread l10_04.cpp
gdb ./a.out
b l10_04.cpp:15
r
info threads
```

```
(gdb) info threads
  Id   Target Id         Frame
  1    Thread 0x7ffff7fd4740 (LWP 27999) "a.out" clone
()
    at ../sysdeps/unix/sysv/linux/x86_64/clone.S:81
* 2    Thread 0x7ffff6f4e700 (LWP 28003) "a.out"
Wallet::addMoney (this=0x7fffffffdc00, money=1000)
    at l10_04.cpp:15  ……
```