Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Образовательный комплекс

Введение в методы параллельного программирования

Раздел 12.

Решение дифференциальных уравнений в частных производных

Гергель В.П., профессор, д.т.н.
Кафедра математического обеспечения ЭВМ

Содержание…

- Постановка задачи
- Методы решения дифференциальных уравнений в частных производных
- Организация параллельных вычислений для систем с общей памятью
  - Проблема блокировки при взаимоисключении
  - Проблема неоднозначности вычислений в параллельных программах
  - Состязание потоков
  - Проблема взаимоблокировки
  - Разрешение тупиков
  - Исключение неоднозначности вычислений
  - Волновые схемы параллельных вычислений
  - Блочное представление данных
  - Балансировка вычислительной нагрузки процессоров

Содержание

- Организация параллельных вычислений для систем с распределенной памятью
  - Разделение данных
  - Ленточная схема разделения данных
  - Параллельное выполнение операций передачи данных
  - Коллективные операции обмена информацией
  - Блочная схема разделения данных
  - Вычислительный конвейер (множественная волна)
  - Операции передачи данных
- Заключение

Введение

- Дифференциальные уравнения в частных производных широко используется при разработке моделей в самых разных областях науки и техники
- Анализ математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения
- Объем выполняемых при этом вычислений является значительным.

Проблематика численного решения дифференциальных уравнений в частных производных является областью интенсивных исследований

Постановка задачи

В качестве учебного примера рассматривается *проблема численного решения задачи Дирихле для уравнения Пуассона*

$$\begin{cases} \dfrac{\delta^2 u}{\delta x^2} + \dfrac{\delta^2 u}{\delta y^2} = f(x,y), & (x,y) \in D, \\ u(x,y) = g(x,y), & (x,y) \in D^0, \end{cases}$$

$$D = \{(x,y) \in D : 0 \le x, y \le 1\}$$

Последовательные методы решения

Метод конечных разностей

- Область решения представляется в виде дискретного набора (*сетки*) точек (*узлов*)
- Последовательность решений равномерно сходится к решению задачи Дирихле, а погрешность решения имеет порядок h^2

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih,\ y_i = jh,\ \ 0 \le i, j \le N+1, \\ h = 1/(N+1), \end{cases}$$

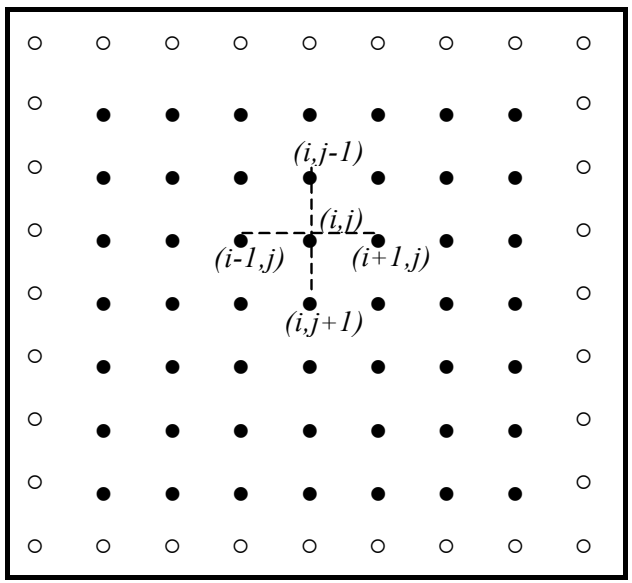
$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}$$

$$u_{ij} = 0.25\,(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij})$$

Итерационные схемы

Метод Гаусса-Зейделя

$$u_{ij}^{k} = 0.25\,(u_{i-1,j}^{k} + u_{i+1,j}^{k-1} + u_{i,j-1}^{k} + u_{i,j+1}^{k-1} - h^2 f_{ij})$$



Трудоемкость

$$T = kmN^2$$

N - число узлов по каждой координате

m - число операций на узел

k - количество итераций

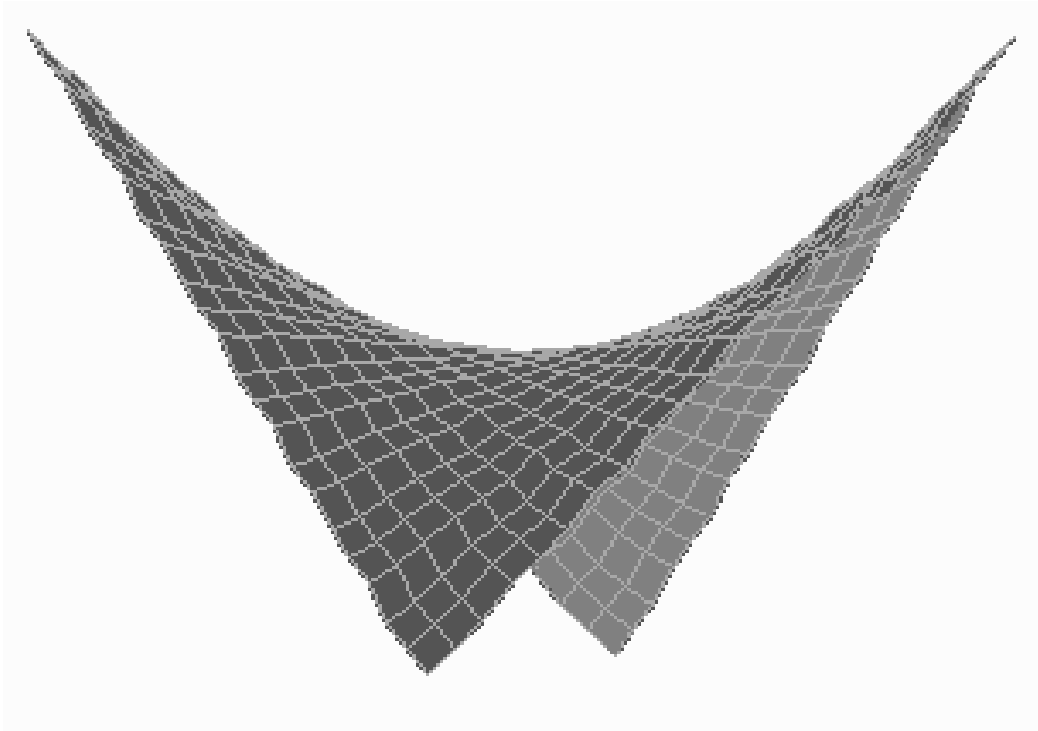
Алгоритм 1: Последовательный алгоритм Гаусса-Зейделя

```
// Алгоритм 12.1
do {
  dmax = 0; // максимальное изменение значений u
  for ( i=1; i<N+1; i++ )
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      dm = fabs(temp-u[i][j]);
      if ( dmax < dm ) dmax = dm;
    }
} while ( dmax > eps );
```

Программа

Пример расчетов

$$\begin{cases} f(x,y) = 0, & (x,y) \in D, \\ 100 - 200\,x, & y = 0, \\ 100 - 200\,y, & x = 0, \\ -100 + 200\,x, & y = 1, \\ -100 + 200\,y, & x = 1, \end{cases}$$



$$N = 100$$
$$\varepsilon = 0.1$$
$$k = 210$$

Организация параллельных вычислений для систем с общей памятью…

- Возможный способ получение ПО для параллельных вычислений – переработка существующих последовательных программ
- Такая переработка выполняется или автоматически компилятором или непосредственно программистом
- Второй подход является преобладающим, т.к. возможности автоматического построения параллельных программ достаточно ограничены
- Использование новых алгоритмических языков параллельного программирования приводит к необходимости значительной переработки существующего программного обеспечения

Организация параллельных вычислений для систем с общей памятью

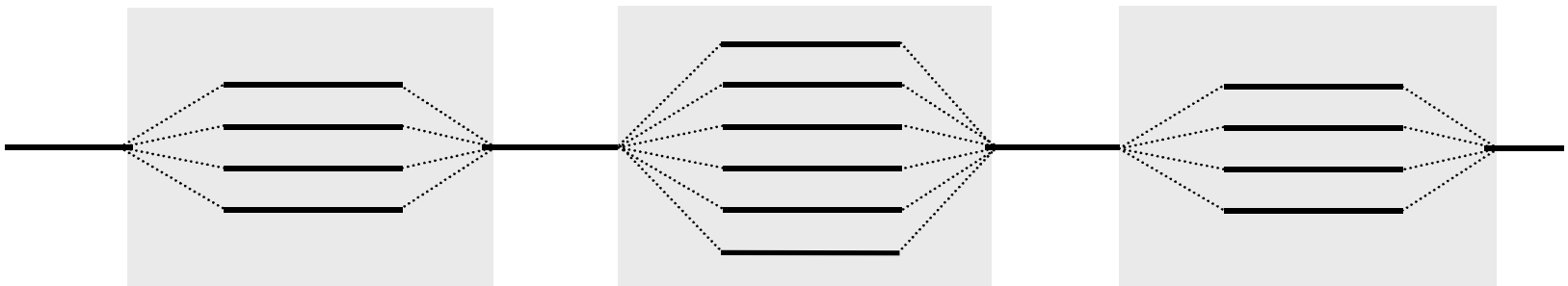
- Возможный решение проблемы – использование тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы
- Директивы дают указания на возможные способы распараллеливания программы, при этом исходный текст программы остается неизменным (!!!)
- Препроцессор при обработке текста программы заменяет директивы параллелизма на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки)
- При отсутствии препроцессора компилятор построит исходный последовательный программный код (!!!)

Единство программного кода для последовательных и параллельных вычислений снижает сложность развития и сопровождения программ

Определение параллелизма при помощи директив позволяет осуществлять поэтапную разработку параллельных программ

Технология OpenMP

- Для организации параллельных вычислений программистом в программу добавляются указания в виде директив (C/C++) или комментариев (Fortran)
- Вилочный (fork-join) –*пульсирующий* - параллелизм - выделение в программе параллельных областей



В результате такого подхода программа представляется в виде набора последовательных (однопотоковых) и параллельных (многопотоковых) участков программного кода

Алгоритм 1.2: Первый вариант параллельного алгоритма Гаусса-Зейделя

```
//Алгоритм 12.2
omp_lock_t dmax_lock;
omp_init_lock (dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax) private(i,temp,d)
  for ( i=1; i<N+1; i++ ) {
#pragma omp parallel for shared(u,n,dmax) private(j,temp,d)
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      omp_set_lock(dmax_lock);
        if ( dmax < d ) dmax = d;
      omp_unset_lock(dmax_lock);
    } // конец вложенной параллельной области
  } // конец внешней параллельной области
} while ( dmax > eps );
```

Программа

Результаты вычислительных экспериментов

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 12.1)		Параллельный алгоритм 12.2		
	k	***T***	***k***	***T***	***S***
100	210	0,06	210	1,97	0,03
200	273	0,34	273	11,22	0,03
300	305	0,88	305	29,09	0,03
400	318	3,78	318	54,20	0,07
500	343	6,00	343	85,84	0,07
600	336	8,81	336	126,38	0,07
700	344	12,11	344	178,30	0,07
800	343	16,41	343	234,70	0,07
900	358	20,61	358	295,03	0,07
1000	351	25,59	351	366,16	0,07
2000	367	106,75	367	1585,84	0,07
3000	370	243,00	370	3598,53	0,07

Оценка подхода

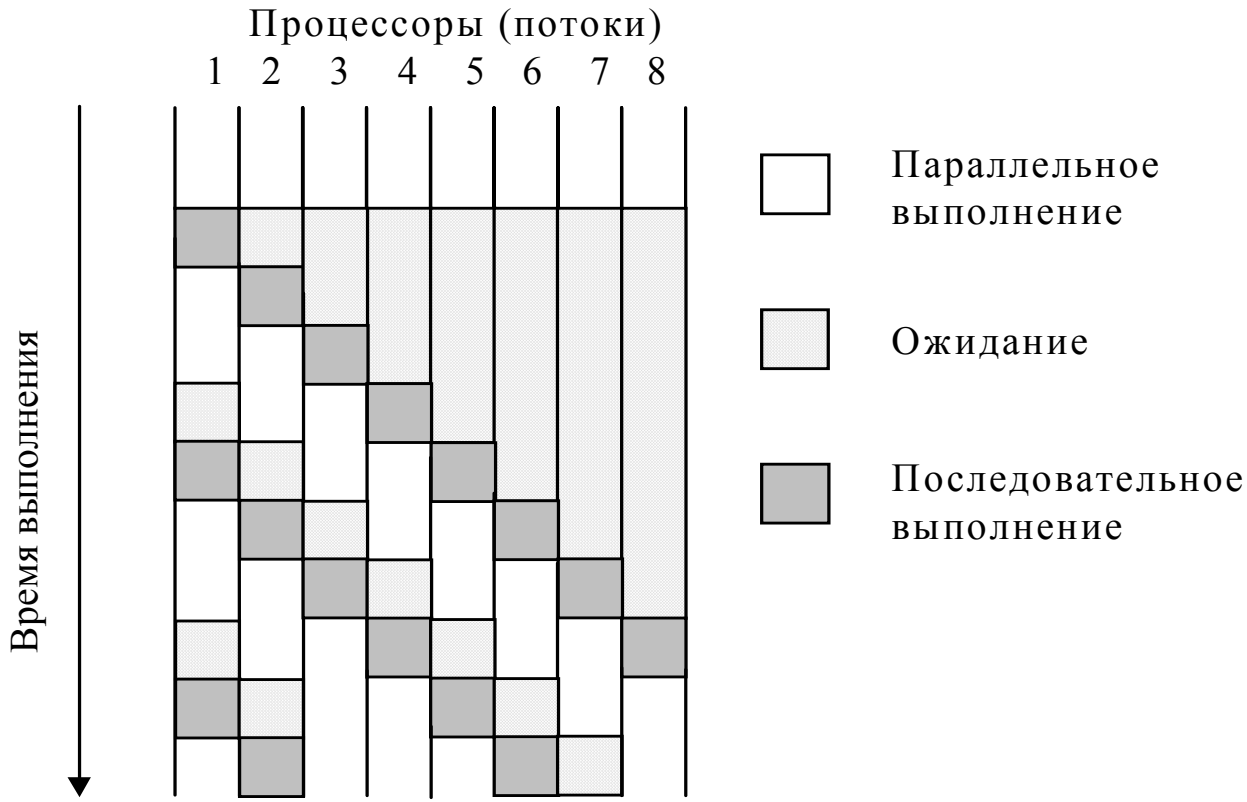
- Разработанный параллельный алгоритм обеспечивает решение поставленной задачи
- Может быть задействовано до N^2 процессоров.
- Чрезмерно высокая *синхронизация* параллельных участков программы
- Слабая загрузка процессоров

Низкая эффективность

Проблема: блокировки при взаимоисключении

- Каждый параллельный поток после усреднения значений должен проверить значение величины *dmax*
- Разрешение на использование переменной получает один поток, другие в это время блокируются. После освобождения общей переменной управление может получить следующий поток и т.д.

Проблема: блокировки при взаимоисключении



В результате многопотоковая параллельная программа превращается в последовательно выполняемый код

Алгоритм 1.3: Второй вариант параллельного алгоритма Гаусса-Зейделя

```
//Алгоритм 12.3
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax)private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
      d = fabs(temp-u[i][j]);
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // конец параллельной области
} while ( dmax > eps );
```

Программа

Результаты вычислительных экспериментов

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 12.1)		Параллельный алгоритм 12.2			Параллельный алгоритм 12.3		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	1,97	0,03	210	0,03	2,03
200	273	0,34	273	11,22	0,03	273	0,14	2,43
300	305	0,88	305	29,09	0,03	305	0,36	2,43
400	318	3,78	318	54,20	0,07	318	0,64	5,90
500	343	6,00	343	85,84	0,07	343	1,06	5,64
600	336	8,81	336	126,38	0,07	336	1,50	5,88
700	344	12,11	344	178,30	0,07	344	2,42	5,00
800	343	16,41	343	234,70	0,07	343	8,08	2,03
900	358	20,61	358	295,03	0,07	358	11,03	1,87
1000	351	25,59	351	366,16	0,07	351	13,69	1,87
2000	367	106,75	367	1585,84	0,07	367	56,63	1,89
3000	370	243,00	370	3598,53	0,07	370	128,66	1,89

Оценка подхода

- Существенное снижение обращений к общей переменной
- Снижение показателя максимально возможного параллелизма до *N*
- Как результат существенное снижению затрат на синхронизацию потоков и уменьшению проявления эффекта сериализации вычислений.

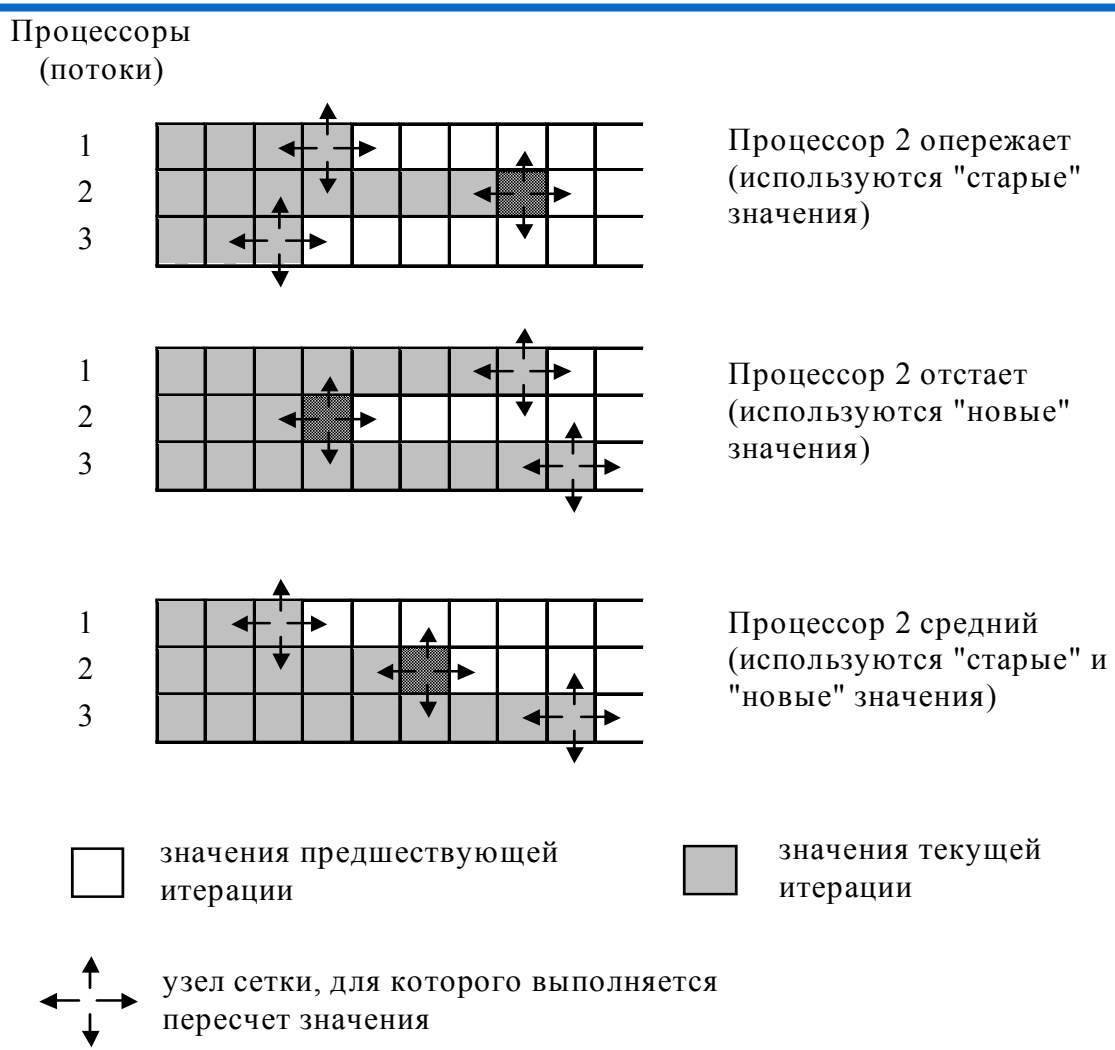
Лучшие показатели ускорения

Возможность неоднозначности вычислений в параллельных программах

- При вычислениях последовательность обработки данных может различаться при разных запусках программы
- Взаиморасположение потоков по области расчетов может быть различным: одни потоки могут опережать другие и, обратно, часть потоков могут отставать
- Причина - *состязанием потоков* (*race condition*)

Временная динамика выполнения параллельных потоков не должна учитываться при разработке параллельных алгоритмов и программ

Состязание потоков

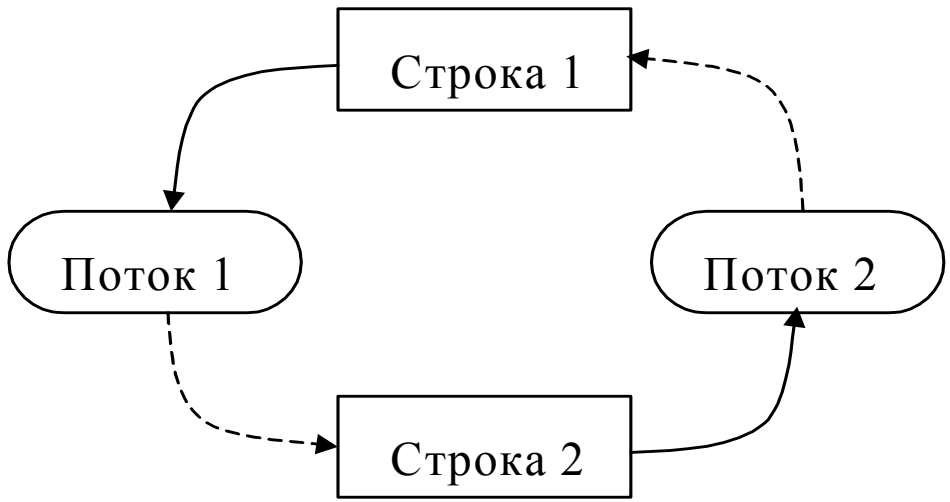


Выход: захват и блокировка используемых строк

Проблема: взаимоблокировка

- Для ограничения доступа к узлам сетки можно ввести набор семафоров **row_lock[N]**, который позволит потокам закрывать доступ к "своим" строкам сетки

```
// поток обрабатывает i строку сетки
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i-1]);
// обработка i строку сетки
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i-1]);
```



Потоки блокируют сначала строки 1 и 2 и только затем переходят к блокировке оставшихся строк - Тупик (!!!)

Разрешение тупиков

Решение: соблюдение строгой последовательности блокировки строк

```
// поток обрабатывает i строку сетки
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i-1]);
// <обработка i строку сетки>
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i-1]);
```

Однозначность вычислений не обеспечивается

Исключение неоднозначности вычислений

- Для исключения неоднозначности вычислений применяют способ, который состоит в разделение места хранения результатов вычислений на предыдущей и текущей итерациях метода сеток (*метод Гаусса-Якоби*)

Алгоритм 1.4: Параллельная реализация сеточного метода Гаусса-Якоби

```
//Алгоритм 12.4
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax) \
                         private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      un[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                 u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-un[i][j])
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // конец параллельной области
  for ( i=1; i<N+1; i++ ) // обновление данных
    for ( j=1; j<N+1; j++ )
      u[i][j] = un[i][j];
} while ( dmax > eps );
```

Программа

Результаты вычислительных экспериментов

Размер сетки	Последовательный метод Гаусса-Якоби (алгоритм 12.4)		Параллельный метод, разработанный по аналогии с алгоритмом 12.3		
	k	*t*	*k*	*T*	*S*
100	5257	1,39	5257	0,73	1,90
200	23067	23,84	23067	11,00	2,17
300	26961	226,23	26961	29,00	7,80
400	34377	562,94	34377	66,25	8,50
500	56941	1330,39	56941	191,95	6,93
600	114342	3815,36	114342	2247,95	1,70
700	64433	2927,88	64433	1699,19	1,72
800	87099	5467,64	87099	2751,73	1,99
900	286188	22759,36	286188	11776,09	1,93
1000	152657	14258,38	152657	7397,60	1,93
2000	337809	134140,64	337809	70312,45	1,91
3000	655210	247726,69	655210	129752,13	1,91

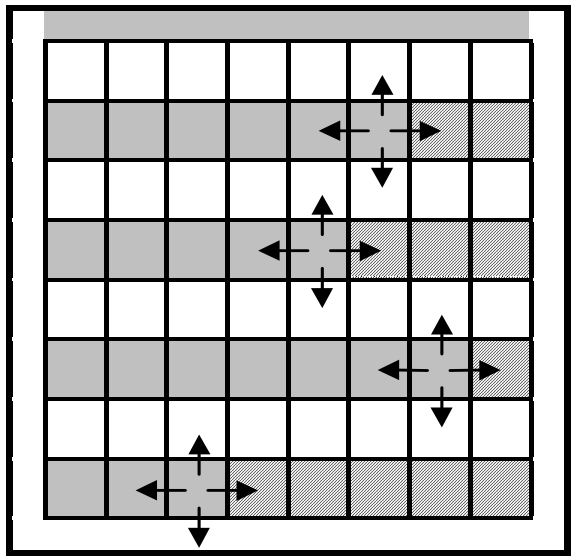
Оценка подхода

- Однозначность вычислений
- Использование дополнительной памяти
- Меньшая скорость сходимости

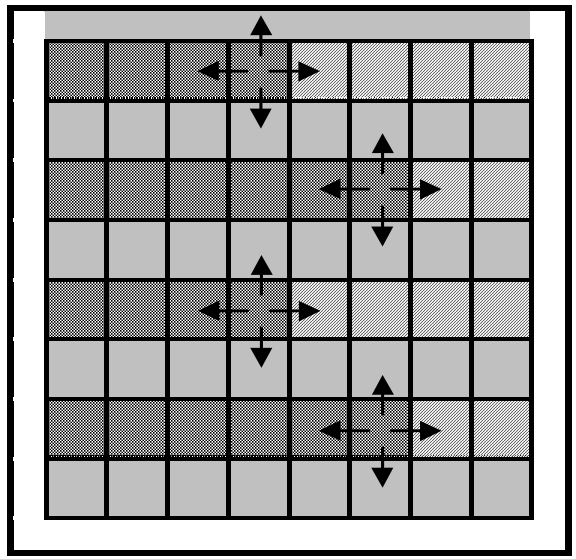
Возможный иной способ устранения взаимозависимости параллельных потоков состоит в использовании *схемы чередования обработки строк*, при которой выполнение каждой итерации подразделяется на два последовательных этапа:

- На первом этапе обрабатываются строки только с четными номерами,
- На втором этапе - строки с нечетными номерами

Схема чередования обработки строк



Этап 1



Этап 2

граничные значения

значения предшествующей итерации

значения после этапа 1 текущей итерации

значения после этапа 2 текущей итерации

Оценка подхода

- Не требуется дополнительной памяти
- Алгоритм обеспечивает однозначность вычислений, но не совпадающие с результатами последовательных расчетов
- Меньшая скорость сходимости

Возможность повышения эффективности расчетов

Оценка подхода

Метод Гаусса-Якоби

Использование дополнительной памяти

Схема чередования обработки строк

Не требуется дополнительной памяти

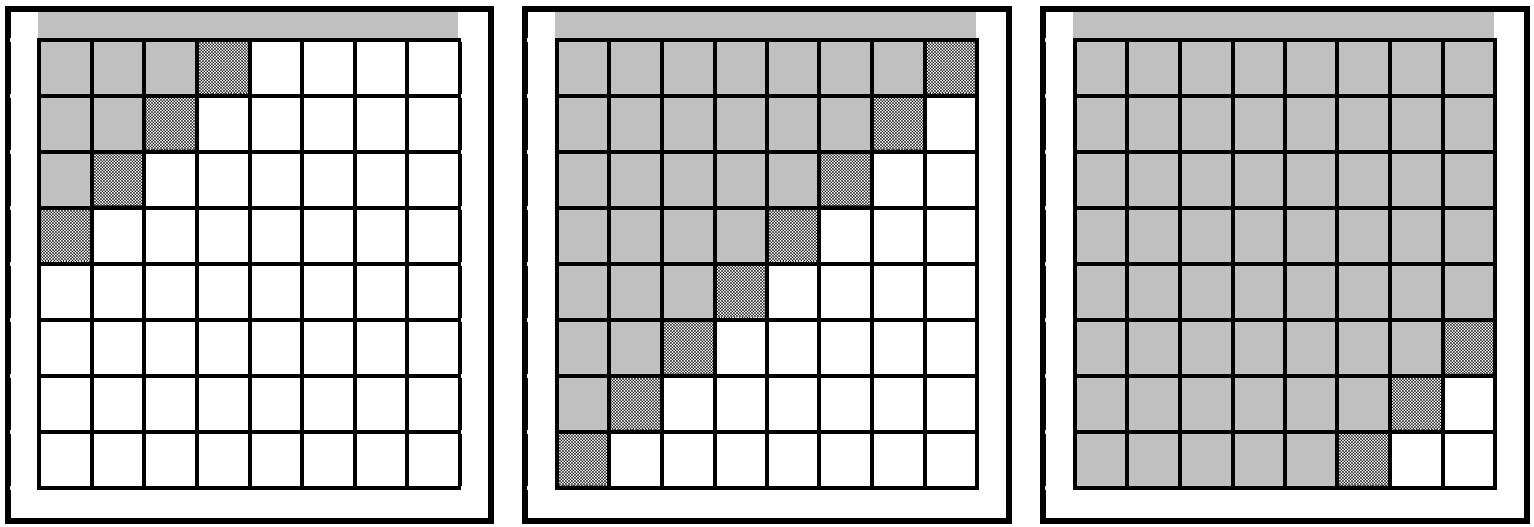
- Алгоритмы обеспечивают однозначность вычислений, но получаемые решения могут не совпадать с результатами последовательных расчетов
- Вычислительные схемы имеют меньшую скорость сходимости, чем исходный вариант метода Гаусса-Зейделя

Волновые схемы параллельных вычислений

- Рассмотрим параллельные алгоритмы обладающими следующими свойствами:
  - выполняемые вычислительные действия, что и исходный последовательный метод
  - полученные решения совпадали с решением исходной вычислительной задачи.
- Один из таких методов - метод волновой обработки данных
- Вычислительная схема состоит в выполнение итерации метода сеток которые разбиваются на последовательность шагов

Волновые схемы параллельных вычислений

- На каждом шаге к вычислениям окажутся подготовленными узлы вспомогательной диагонали сетки с номером, определяемом номером этапа



Нарастание волны Пик волны Затухание волны

граничные значения

значения текущей итерации

значения предшествующей итерации

узлы, в которых могут быть пересчитаны значения

Алгоритм 1.5: Параллельный алгоритм реализующий волновую схему вычислений…

```
//Алгоритм 12.5
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
  // нарастание волны (nx – размер волны)
  for ( nx=1; nx<N+1; nx++ ) {
    dm[nx] = 0;
#pragma omp parallel for shared(u,nx,dm) private(i,j,temp,d)
    for ( i=1; i<nx+1; i++ ) {
      j    = nx + 1 – i;
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm[i] < d ) dm[i] = d;
    } // конец параллельной области
  }
```

Алгоритм 1.5: Параллельный алгоритм реализующий волновую схему вычислений

```
// затухание волны
for ( nx=N-1; nx>0; nx-- ) {
#pragma omp parallel for shared(u,nx,dm) private(i,j,temp,d)
    for ( i=N-nx+1; i<N+1; i++ ) {
      j    = 2*N - nx – I + 1;
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm[i] < d ) dm[i] = d;
    } // конец параллельной области
  }
#pragma omp parallel for shared(n,dm,dmax) private(i)
  for ( i=1; i<nx+1; i++ ) {
    omp_set_lock(dmax_lock);
      if ( dmax < dm[i] ) dmax = dm[i];
    omp_unset_lock(dmax_lock);
  } // конец параллельной области
} while ( dmax > eps );
```

Программа

Волновые схемы параллельных вычислений

- Последняя часть расчетов для определения максимальной погрешности неэффективна из-за высоких затрат на синхронизацию
- *Фрагментирование* (*chucking*) - уменьшения синхронизации за счет увеличения размера последовательных участков
- Возможный вариант реализации такого подхода может состоять в следующем:

```
chunk = 200; // размер последовательного участка
#pragma omp parallel for shared(n,dm,dmax)private(i,d)
  for ( i=1; i<nx+1; i+=chunk ) {
    d = 0;
    for ( j=i; j<i+chunk; j++ )
      if ( d < dm[j] ) d = dm[j];
    omp_set_lock(dmax_lock);
      if ( dmax < d ) dmax = d;
    omp_unset_lock(dmax_lock); } // конец параллельной области
```

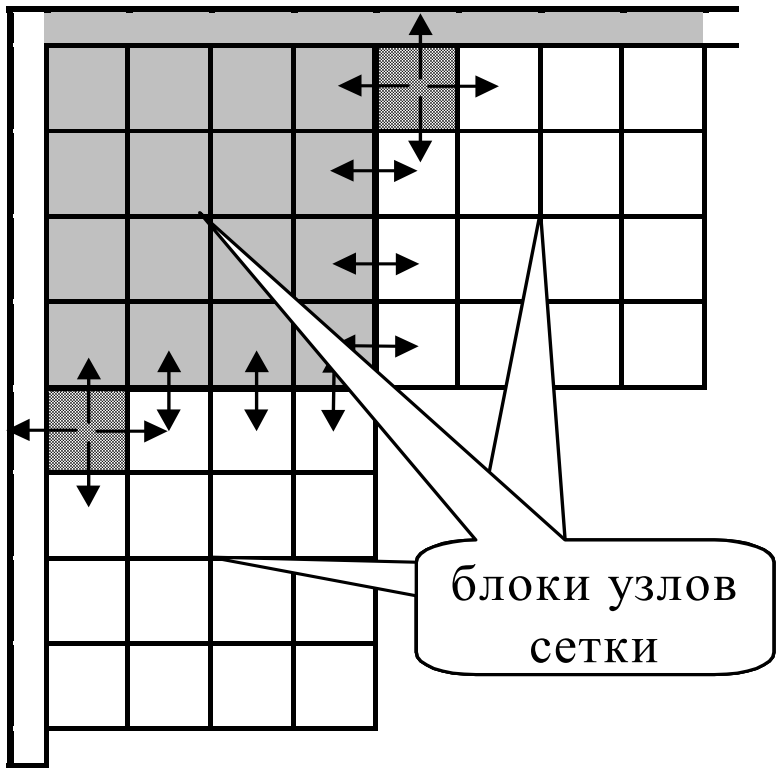
Результаты экспериментов

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 1)		Параллельный алгоритм 12.5		
	k	***t***	***k***	***T***	***S***
100	210	0,06	210	0,30	0,21
200	273	0,34	273	0,86	0,40
300	305	0,88	305	1,63	0,54
400	318	3,78	318	2,50	1,51
500	343	6,00	343	3,53	1,70
600	336	8,81	336	5,20	1,69
700	344	12,11	344	8,13	1,49
800	343	16,41	343	12,08	1,36
900	358	20,61	358	14,98	1,38
1000	351	25,59	351	18,27	1,40
2000	367	106,75	367	69,08	1,55
3000	370	243,00	370	149,36	1,63

Оценка подхода

- Неэффективность использования кэша
- Для повышения быстродействия вычисления за счет кэша необходимо следующее:
  - выполняемые вычисления использовали одни и те же данные многократно (*локальность обработки* данных),
  - выполняемые вычисления осуществляли доступ к элементам памяти с последовательно возрастающими адресами (*последовательность доступа*)
- Для эффективного использования кэша в качестве распределяемых между процессорами действий процедуру обработки некоторой прямоугольной подобласти (*блока*) сетки области расчетов

Блочное представление данных



граничные значения

значения предшествующей итерации

значения текущей итерации

узлы, в которых могут быть пересчитаны значения

значения, которые должны передаваться между границами блоков

Алгоритм 1.6: Блочный подход к методу волновой обработки данных

```
//Алгоритм 12.6
do {
  // нарастание волны (размер волны равен nx+1)
  for ( nx=0; nx<NB; nx++ ) { // NB количество блоков
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = nx – i;
      // <обработка блока с координатами (i,j)>
    } // конец параллельной области
  }
  // затухание волны
  for ( nx=NB-2; nx>-1; nx-- ) {
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = 2*(NB-1) - nx – i;
      // <обработка блока с координатами (i,j)>
    } // конец параллельной области
  }
  // <определение погрешности вычислений>
} while ( dmax > eps );
```

Программа

Результаты вычислительных экспериментов

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 12.1)		Параллельный алгоритм 12.5			Параллельный алгоритм 12.6		
	K	***T***	***k***	***t***	***S***	***k***	***t***	***S***
100	210	0,06	210	0,30	0,21	210	0,16	0,40
200	273	0,34	273	0,86	0,40	273	0,59	0,58
300	305	0,88	305	1,63	0,54	305	1,53	0,57
400	318	3,78	318	2,50	1,51	318	2,36	1,60
500	343	6,00	343	3,53	1,70	343	4,03	1,49
600	336	8,81	336	5,20	1,69	336	5,34	1,65
700	344	12,11	344	8,13	1,49	344	10,00	1,21
800	343	16,41	343	12,08	1,36	343	12,64	1,30
900	358	20,61	358	14,98	1,38	358	15,59	1,32
1000	351	25,59	351	18,27	1,40	351	19,30	1,33
2000	367	106,75	367	69,08	1,55	367	65,72	1,62
3000	370	243,00	370	149,36	1,63	370	140,89	1,72

Оценка подхода

- Обработка блоков выполняется на разных процессорах и блоки не пересекаются по данным, тогда будут отсутствовать накладные расходы для обеспечения однозначности (когерентности) кэшей разных процессоров.
- Возможность простоев процессоров

Возможность повышения эффективности расчетов

Балансировка вычислительной нагрузки процессоров

- Размер блока определяет *степень разбиения* (*granularity*) вычислений для распараллеливания
- Подбирая значение степени разбиения можно управлять эффективностью параллельных вычислений
- Для обеспечения равномерности (*балансировки*) загрузки процессоров - вычислительные действия организуются в виде *очереди заданий*
- В ходе вычислений освободившийся процессор может запросить для себя работу из этой очереди

Очередь заданий является *общим подходом организации параллельных вычислений* для систем с общей памятью

Алгоритм 1.7:Общая схема балансировки вычислений с использованием очереди

```
//Алгоритм 12.7
// <инициализация служебных данных>
// <загрузка в очередь указателя на начальный блок>
// взять блок из очереди (если очередь не пуста)
while ( (pBlock=GetBlock()) != NULL ) {
  // <обработка блока>
  // отметка готовности соседних блоков
  omp_set_lock(pBlock->pNext.Lock); // сосед справа
    pBlock->pNext.Count++;
    if ( pBlock->pNext.Count == 2 )
      PutBlock(pBlock->pNext);
  omp_unset_lock(pBlock->pNext.Lock);
  omp_set_lock(pBlock->pDown.Lock); // сосед снизу
    pBlock->pDown.Count++;
    if ( pBlock->pDown.Count == 2 )
      PutBlock(pBlock->pDown);
  omp_unset_lock(pBlock->pDown.Lock);
} // завершение вычислений, т.к. очередь пуста
```

Организация параллельных вычислений для систем с распределенной памятью

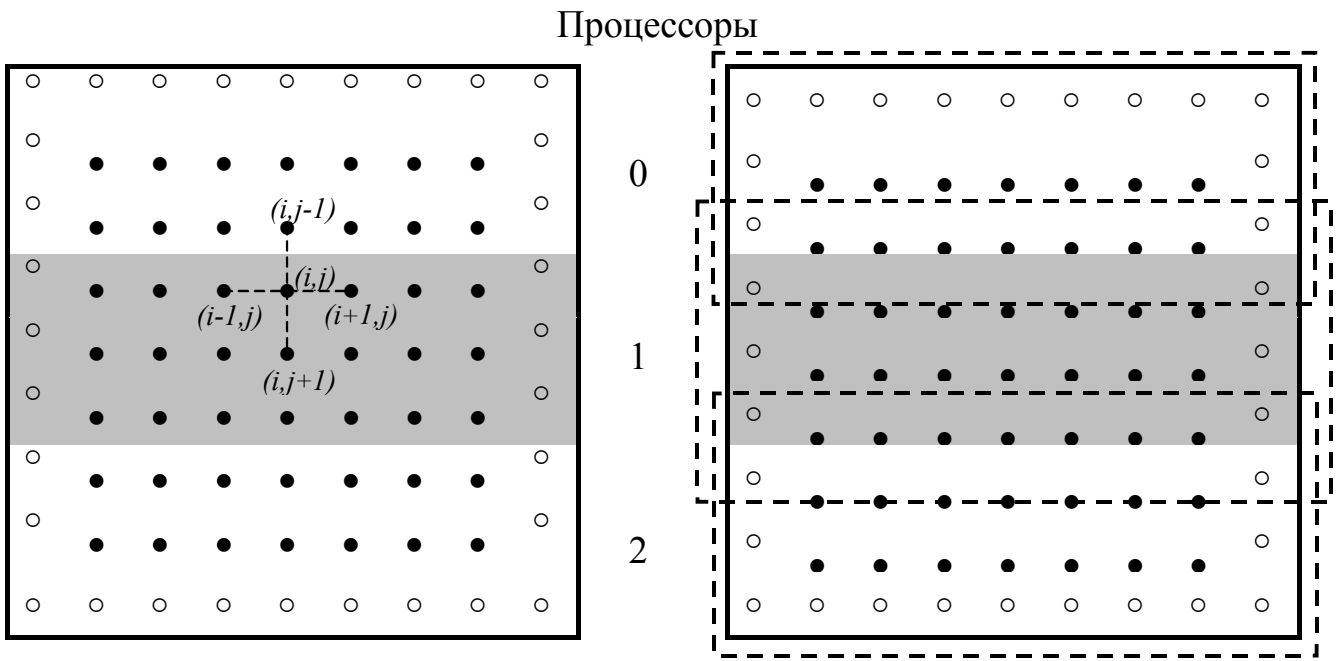
- Многие проблемы параллельного программирования (*состязание вычислений*, *тупики*, *сериализация*) являются общими для систем с общей и распределенной памятью
- Взаимодействие параллельных участков программы на разных процессорах может быть обеспечено только при помощи *передачи сообщений* (*message passing*)

Разделение данных

- В рассматриваемой учебной задаче по решению задачи Дирихле возможны два различных способа разделения данных:
  - *одномерная* или *ленточная* схема разбиения вычислительной сетки,
  - *двухмерное* или *блочное* разбиение вычислительной сетки.
- При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы
- Число полос определяется количеством процессоров, размер полос обычно является одинаковым
- Полосы для обработки распределяются между процессорам

Ленточная схема разделения данных

- Важно отметить следующее:
  - процессор, выполняющий обработку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки,
  - дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток



Алгоритм 1.8: Схема Гаусса-Зейделя, ленточное разделение данных

```
//Алгоритм 12.8
// схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // <обмен граничных строк полос с соседями>
  // <обработка полосы>
  // <вычисление общей погрешности вычислений dmax>}
 while ( dmax > eps ); // eps - точность решения
```

Программа

Схема обмена данными между процессорами

- На первом этапе каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора.

- На втором этапе каждый процессор передает свою верхнюю граничную строку своим предыдущим соседям и принимает переданные строки от следующих процессоров.

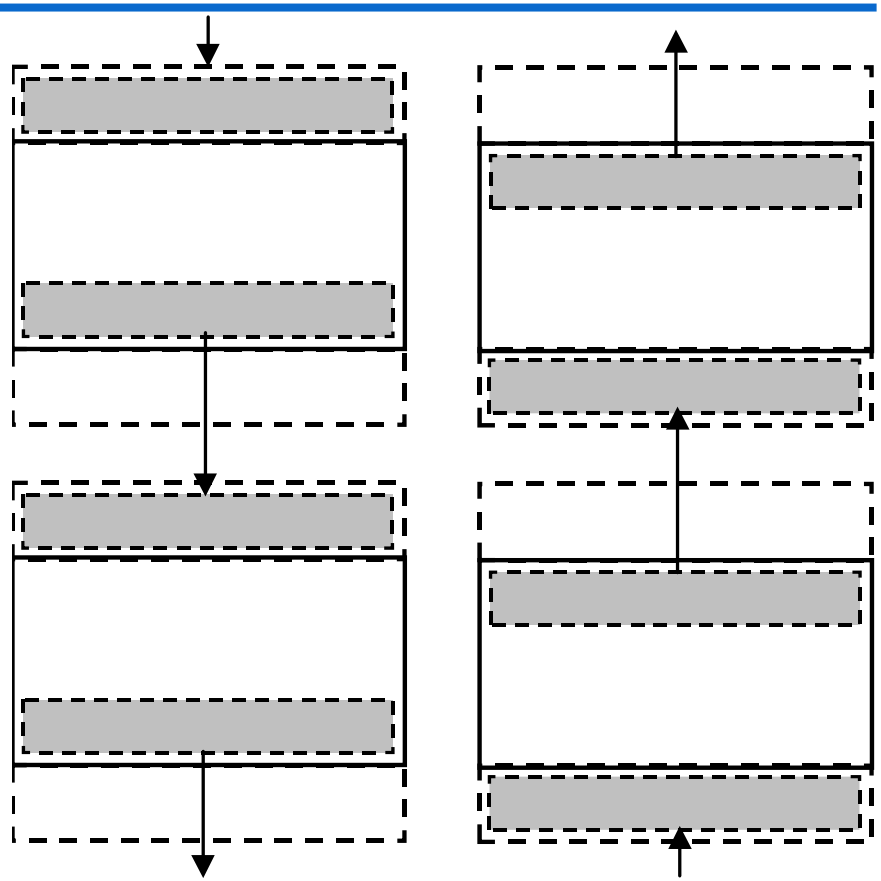


Схема обмена данными между процессорами

- Выполнение операций передачи данных в общем виде может быть представлено следующим образом:

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
```

- Для передачи данных могут быть задействованы два различных механизма – *блокирующая* и *неблокирующая* передача
- Оба эти варианта операций передачи широко используются при организации параллельных вычислений и имеют свои достоинства и свои недостатки

Параллельное выполнение операций передачи данных

- На первом шаге все процессоры с нечетными номерами отправляют данные, а процессоры с четными номерами осуществляют прием этих данных
- На втором шаге роли процессоров меняются – четные процессоры выполняют *Send*, нечетные процессоры исполняют операцию приема *Receive*

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum % 2 == 1 ) { //  нечетный процессор
  if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
  if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
}
else { //  процессор с четным номером
  if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
  if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
}
```

Коллективные операции обмена информацией

- Выполнение операций сборки и рассылки данных может быть реализовано с использованием *каскадной схемы* обработки данных
- Получение максимального значения локальных погрешностей, вычисленных на каждом процессоре
  - предварительного нахождения максимальных значений для отдельных пар процессоров (данные вычисления могут выполняться параллельно),
  - попарный поиск максимума среди полученных результатов и т.д.
- Общее количество параллельных итераций по каскадной схеме $log_2 P$ для получения конечного значения (P – количество процессоров).

Алгоритм 1.8: Уточненный вариант Гаусса-Зейделя, ленточное разделение данных

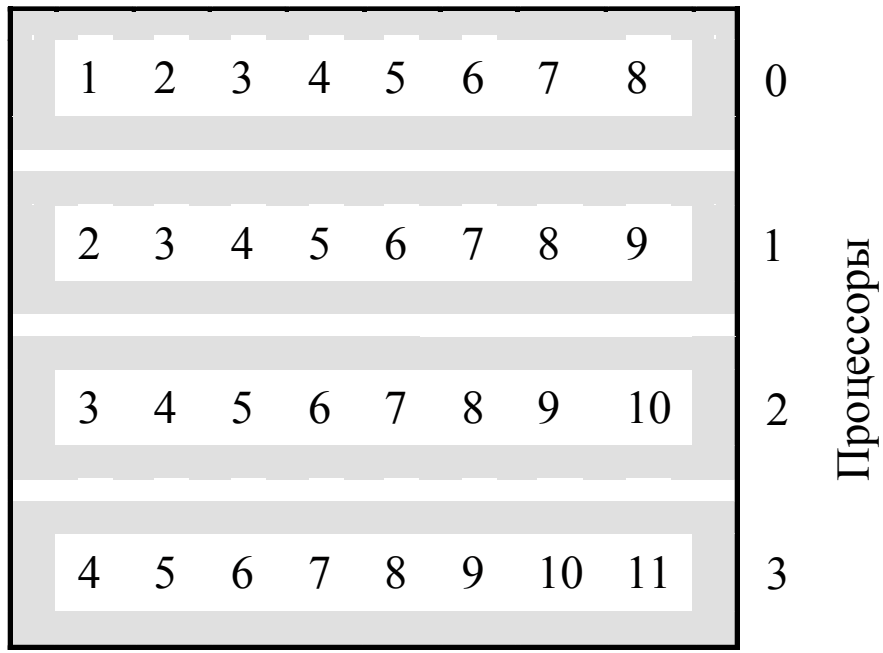
```
//Алгоритм 12.8 – уточненный вариант
// схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // обмен граничных строк полос с соседями
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc
);
  // <обработка полосы с оценкой погрешности dm>
  // вычисление общей погрешности вычислений dmax
  Reduce(dm,dmax,MAX,0);
  Broadcast(dmax,0);
} while ( dmax > eps ); // eps - точность решения
```

Результаты вычислительных экспериментов

Размер сетки	Последовательный метод Гаусса-Зейделя		Параллельный алгоритм 1.8		
	k	***t***	***k***	***t***	***S***
100	210	0,06	210	0,54	0,11
200	273	0,35	273	0,86	0,41
300	305	0,92	305	0,92	1,00
400	318	1,69	318	1,27	1,33
500	343	2,88	343	1,72	1,68
600	336	4,04	336	2,16	1,87
700	344	5,68	344	2,52	2,25
800	343	7,37	343	3,32	2,22
900	358	9,94	358	4,12	2,41
1000	351	11,87	351	4,43	2,68
2000	367	50,19	367	15,13	3,32
3000	364	113,17	364	37,96	2,98

Волновые вычисления при ленточной схеме разделения данных

- Для образования волны вычислений представим логически каждую полосу узлов области расчетов в виде набора блоков и организуем обработку полос поблочно в последовательном порядке



Блочная схема разделения данных

- При блочном представлении данных увеличивается количество граничных строк на каждом процессоре, что приводит, к большему числу операций передачи данных при обмене граничных строк
- Блочная схема представления области расчетов становится оправданной при большом количество узлов сетки области расчетов

Блочная схема разделения данных

```
//Алгоритм 12.9
// схема Гаусса-Зейделя, блочное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // получение граничных узлов
  if ( ProcNum / NB != 0 ) { // строка не нулевая
    // получение данных от верхнего процессора
    Receive(u[0][*],M+2,TopProc); // верхняя строка
    Receive(dmax,1,TopProc);      // погрешность
  }
  if ( ProcNum % NB != 0 ) { // столбец не нулевой
    // получение данных от левого процессора
    Receive(u[*][0],M+2,LeftProc); // левый столбец
    Receive(dm,1,LeftProc);        // погрешность
    If ( dm > dmax ) dmax = dm;
  }
```

Блочная схема разделения данных

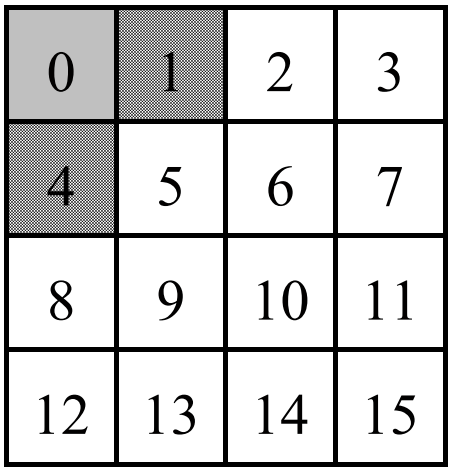
```
 // <обработка блока с оценкой погрешности dmax>
  // пересылка граничных узлов
  if ( ProcNum / NB != NB-1 ) { // строка решетки не
                                // последняя
    // пересылка данных нижнему процессору
    Send(u[M+1][*],M+2,DownProc); // нижняя строка
    Send(dmax,1,DownProc);        // погрешность
  }
  if ( ProcNum % NB != NB-1 ) { // столбец решетки
                                // не последний
    // пересылка данных правому процессору
    Send(u[*][M+1],M+2,RightProc); // правый столбец
    Send(dmax,1, RightProc);       // погрешность
  }
  // синхронизация и рассылка погрешности dmax
  Barrier();
  Broadcast(dmax,NP-1);
} while ( dmax > eps ); // eps - точность решения
```

Программа

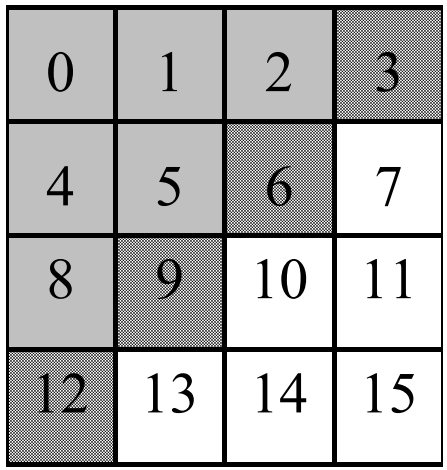
Вычислительный конвейер (множественная волна)

- Эффективность организации волновых вычислений снижается для процессоров, которые занимаются обработкой данных только в моменты, когда их блоки попадают во фронт волны вычислений
- Для улучшения балансировки вычислительной нагрузки между процессорами применяют организацию *множественной волны вычислений*
- Идея организации состоит в следующем - процессоры после отработки волны текущей итерации расчетов могут приступить к выполнению волны следующей итерации метода сеток

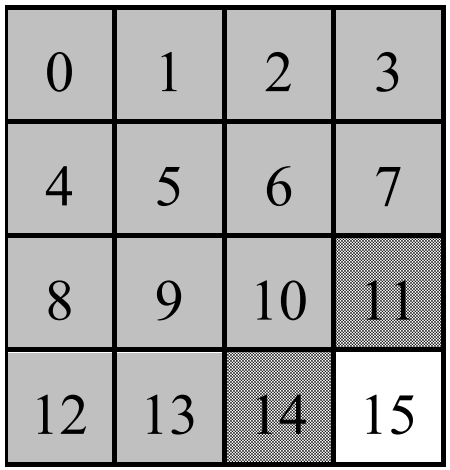
Вычислительный конвейер (множественная волна)



Нарастание волны



Пик волны



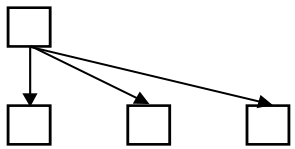
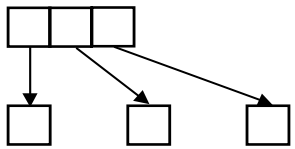
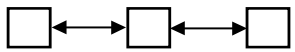
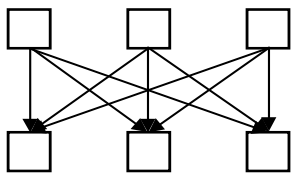
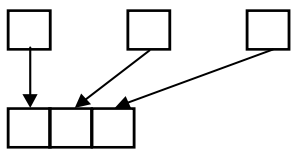
Затухание волны

блоки со значениями текущей итерации

блоки, в которых могут быть пересчитаны значения

блоки со значениями предшествующей итерации

Операции передачи данных

Операция приема-передачи	процессоры	Обеспечивающая процедура MPI
	0 1 2	
Рассылка количества узлов сетки		MPI_Bcast
Рассылка полос или блоков узлов сетки		MPI_Scatter
Обмен границ соседних полос или блоков		MPI_Sendrecv
Сборка и рассылка погрешности вычислений		MPI_Allreduce
Сборка полос или блоков узлов сетки		MPI_Gather

Заключение

- Рассмотрены способы построения параллельных алгоритмов для систем с общей и разделяемой памяти на примере решения дифференциальных уравнений в частных производных
- При изложении организации параллельных вычислений для систем с общей памятью основное внимание уделяется технологии OpenMP, также приводятся проблемы, возникающие при применении этой технологии и решения этих проблем
- При изложении организации параллельных вычислений для систем с распределенной памятью основное внимание уделяется разделению данных и обмену информацией между процессорами

Заключение

- Рассматриваются различные механизмы приема - передачи данных, такие как синхронные и асинхронные
- Теоретические оценки позволяют достаточно точно определить показатели эффективности параллельных вычислений

Вопросы для обсуждения

- Как повысить эффективность методов волновой обработки данных?
- Как очередь заданий позволяет балансировать нагрузку процессорам?
- Какие проблемы приходится решать при организации параллельных вычислений на системах с распределенной памяти?
- Какие основные операции передачи данных используются в параллельных методах решения задачи Дирихле ?

Темы заданий для самостоятельной работы

- Выполните реализацию параллельного алгоритма реализующий волновую схему вычислений и параллельный метод, в котором реализуется блочный подход к методу волновой обработки данных
- Постройте теоретические оценки времени работы этих алгоритмов с учетом параметров используемой вычислительной системы
- Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с полученными теоретическими оценками

Литература

- Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем – СПб.: БХВ-Петербург,2002.
- Березин И.С., Жидков И.П. Методы вычислений.-М.:Наука,1966
- Корнеев В.В. Параллельное программирование в MPI. Москва-Ижевск: Институт компьютерных исследований,2003
- П. Тихонов А.Н., Самарский А.А. Уравнения математической физики. – М.: Наука,1977
- Хамахер К., Вранешич З., Заки С. Организация ЭВМ. –СПб:Питер,2003
- Fox G.C. et al. Solving Problems on Concurrent Processors.- Prentice Hall, Englewood Cliffs, NJ,1988
- Group W,Lusk E, Skjellum A. Using MPI. Portable Parallel Programming with the Message-Passing Interface. –MIT Press,1994
- Pfister, G.P. In Search of Clusters. Prentice Hall PTR, Upper Saddle River, NJ 1995.
- Rajkumar Buyya. High Performance Cluster Computing. Volume 1: Architectures and Systems. Volume 2:Programming and Applications. Prentice Hall PTR, Prentice-Hall Inc.,1999
- Roosta, S.H. Parallel Processing and Parallel Algorithms: Theory and Computation. Springer-Verlag,NY.2000.
- Xu, Z., Hwang, K. Scalable Parallel Computing Technology, Architecture, Programming. McGraw-Hill, Boston.1998.

Авторский коллектив

Гергель В.П., профессор, д.т.н., руководитель

Гришагин В.А., доцент, к.ф.м.н.

Абросимова О.Н., ассистент (раздел 10)

Лабутин Д.Ю., ассистент (система ПараЛаб)

Курылев А.Л., ассистент (лабораторные работы 4, 5)

Сысоев А.В., ассистент (раздел 1)

Гергель А.В., аспирант (раздел 12, лабораторная работа 6)

Лабутина А.А., аспирант (разделы 7,8,9, лабораторные работы 1, 2, 3, система ПараЛаб)

Сенин А.В., аспирант (раздел 11, лабораторные работы по Microsoft Compute Cluster)

Ливерко С.В. (система ПараЛаб)

О проекте

Целью проекта является создание образовательного комплекса "Многопроцессорные вычислительные системы и параллельное программирование", обеспечивающий рассмотрение вопросов параллельных вычислений, предусматриваемых рекомендациями Computing Curricula 2001 Международных организаций IEEE-CS и ACM. Данный образовательный комплекс может быть использован для обучения на начальном этапе подготовки специалистов в области информатики, вычислительной техники и информационных технологий.

Образовательный комплекс включает **учебный курс "Введение в методы параллельного программирования" и лабораторный практикум "Методы и технологии разработки параллельных программ"**, что позволяет органично сочетать фундаментальное образование в области программирования и практическое обучение методам разработки масштабного программного обеспечения для решения сложных вычислительно-трудоемких задач на высокопроизводительных вычислительных системах.

Проект выполнялся в Нижегородском государственном университете им. Н.И. Лобачевского на кафедре математического обеспечения ЭВМ факультета вычислительной математики и кибернетики (http://www.software.unn.ac.ru). Выполнение проекта осуществлялось при поддержке компании Microsoft.