**Министерство образования Республики Беларусь**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра вычислительной математики**

БОЛТАЧ

Вадим Юрьевич

**РАЗРАБОТКА И РЕАЛИЗАЦИЯ НА МНОГОЯДЕРНОМ ПРОЦЕССОРЕ БЛОЧНОГО ПАРАЛЛЕЛЬНОГО ИТЕРАЦИОННОГО АЛГОРИТМА ЧИСЛЕННОГО РЕШЕНИЯ ДВУМЕРНОЙ ЗАДАЧИ ДИРИХЛЕ ДЛЯ УРАВНЕНИЯ ПУАССОНА**

Дипломная работа

Научный руководитель:

доктор физ.-мат. наук,

профессор Н.А. Лиходед

Допущен к защите

“\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_ 2016 г

Зав. кафедрой вычислительной математики

кандидат физико-математических наук, доцент П.А. Мандрик

Минск 2016

Оглавление

Глава 1 Параллельные алгоритмы шаблонных вычислений 5

1.1 Итерационные процессы для решения разностной задачи Дирихле 5

1.2 Тайлинг. Тайлинг с перекрытием вычислений 8

1.3 Блочный алгоритм, использующий скошенные циклы 11

1.4 Блочный алгоритм, использующий тайлинг с перекрытиями 14

Глава 2 Базовая архитектура и принципы работы центрального процессора 17

2.1 Кэширование 17

2.2 Принцип работы кэша 18

2.3 Виды промахов 20

2.4 Мультипроцессор и многопоточность 21

2.5 Взаимодействие потоков 22

Глава 3 OpenMP 26

3.1 Функции исполняющей среды 28

3.2 Функции синхронизации/блокировки 29

3.3 Функции работы с таймерами 30

Глава 4 Параллельные алгоритмы для реализации на многоядерных компьютерах с общей памятью 31

4.1 Точечный параллельный алгоритм 31

4.2 Блочный параллельный алгоритм, использующий тайлинг с перекрытиями. 32

4.3 О применимости рассмотренных подходов к распараллеливанию и улучшению локальности 34

4.4 Улучшение локальности последовательного алгоритма 35

Глава 5 Вычислительные эксперименты 36

5.1 Задача 36

5.2 Характеристики вычислительной машины 36

5.3 Оценка погрешности реализованного алгоритма 36

5.4 Оценка прироста производительности при использовании точечного параллельного алгоритма 37

5.5 Зависимость скорости вычислений от размера тайла при плоском и трехмерном с перекрытиями алгоритмах тайлинга 38

5.6 Зависимость скорости вычислений от количества итераций входящих в гало тайл для алгоритма с перекрытиями 39

5.7 Сравнение производительности алгоритма использующего обьемный тайлинг с перекрытиями при оптимальном размере тайла с точечным параллельным алгоритмом 40

Заключение 42

Список использованных источников 43

Приложение А 44

РЕФЕРАТ

Диплом, 50 страниц, 7 рисунков, 5 таблиц, 10 формул, 9 источников.

**Ключевые слова**: ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ, МНОГОЯДЕРНЫЙ ПРОЦЕССОР, ЗАДАЧА ДИРИХЛЕ ДЛЯ УРАВНЕНИЯ ПУАССОНА, БЛОЧНЫЙ ПАРАЛЕЛЛЬНЫЙ АЛГОРИТМ.

**Объект исследования:** блочный параллельный итерационный алгоритм численного решения задачи Дирихле для двумерного уравнения Пуассона.

**Цель работы:** исследование, реализация и анализ эффективности параллельных алгоритмов решения задачи Дирихле для двумерного уравнения Пуассона.

**Методы работы:** тайлинг, методы получения зернистых версий параллельных алгоритмов, вычислительные эксперименты на многоядерном процессоре.

**В результате** проведенной работы были исследован и реализован на многоядерном процессоре блочный параллельный итерационный алгоритм численного решения задачи Дирихле для уравнения Пуассона, и сделан вывод о его эффективности.

# Глава 1 Параллельные алгоритмы шаблонных вычислений

Многие алгоритмы численного решения уравнений математической физики, как и другие вычисления с использованием шаблонов, характеризуются обновлением точек сетки с использованием соседних точек. В этой главе представлены некоторые современные подходы распараллеливания и улучшения локальности широкого класса алгоритмов шаблонных вычислений. Исследуется применение тайлинга (т.е. применение разбиения на макрооперации-тайлы) и аффинных преобразований. Тайлинг и аффинные преобразования являются важнейшими преобразованиями получения параллельных версий алгоритмов для эффективной реализации на многоядерных вычислительных системах различной архитектуры.

В качестве примера для демонстрации способов распараллеливания и улучшения локальности алгоритмов будем рассматривать алгоритм зейделевского типа численного решения двумерной задачи Дирихле для уравнения Пуассона.

## 1.1 Итерационные процессы для решения разностной задачи Дирихле

Рассмотрим двумерную задачу Дирихле для уравнения Пуассона, описывающую стационарное распределение тепла в прямоугольной области *G=*[0<*x*1<*l*1]×[0<*x*2<*l*2] с границей Г:

–*f*(*x*1,*x*2), (*x*1,*x*2)*G*,

*u*(*x*1,*x*2)|Г*=*μ(*x*1,*x*2).

Для численного решения задачи введем в области *G+*Г сетку узлов

{*ih*1*, i=*0,1,…,*Nx*, *Nxh*1*=l*1, *jh*2*, j=*0,1,…,*Ny*, *Nyh*2*=l*2}.

Используя пятиточечный шаблон «крест» для вычисления значений производных и сеточное представление функций, запишем разностную задачу Дирихле:

, *i=*1,…,*Nx–*1, *j=*1,…,*Ny–*1,

*y*|γ*h=*μ(*x*1,*x*2)|γ*h*,

где γ*h* – граничные узлы (кроме угловых узлов, они схемой не используются).

Записанную систему линейных алгебраических уравнений относительно неизвестных *yi,j* обычно решают итерационными методами. Перед началом вычислений нужно задаь начальное (например, нулевое) приближение  во внутренних точках сетки и заполнить значения  в граничных узлах точными значениями μ*i,j*.

Итерационный процесс Якоби имеет следующий вид:

,

*i=*1,…,*Nx–*1, *j=*1,…,*Ny–*1, *l=*0, 1, …

Основная часть псевдокода процесса Якоби (*h*1*=h*2*=h*):

do *l =* 1, *rit* // *rit* – некоторое фиксированное число итераций

do *i =* 1, *Nx–*1 // *Nx–*1 – число строк матрицы, задающей

внутренние узлы сетки

do *j =* 1,*Ny–*1 // *Ny–*1 – число столбцов матрицы, задающей

внутренние узлы сетки

*u*(*i*,*j*)*=*0,25(*low*(*i*–1,*j*)*+low*(*i*,*j–*1)*+low*(*i*,*j+*1)*+low*(*i+*1,*j*))*+h*2*f*(*i*,*j*)

enddo

enddo

do *i =* 1, *Nx–*1

do *j =* 1,*Ny–*1

*low*(*i*,*j*)*=u*(*i*,*j*)

enddo

enddo

enddo

Итерационный процесс Зейделя имеет следующий вид:

,

Основная часть псевдокода процесса Зейделя (*h*1*=h*2*=h*):

do *l =* 1, *rit*

do *i =* 1, *Nx–*1

do *j =* 1,*Ny–*1

*u*(*i*,*j*)*=*0,25(*u*(*i*–1,*j*)+*u*(*i*,*j–*1)+*u*(*i*,*j+*1)+*u*(*i+*1,*j*))*+h*2*f*(*i*,*j*) (1)

enddo

enddo

enddo

Метод Зейделя является частным случаем более употребительного на практике метода верхней релаксации (SOR). Основную часть численного решения двумерной задачи Дирихле для уравнения Пуассона методом верхней релаксации можно представить в следующем виде:

do *l =* 1, *rit* // *rit* – некоторое фиксированное число итераций

do *i =* 1, *Nx–*1 // *Nx–*1 – число строк матрицы, задающей

внутренние узлы сетки

do *j =* 1,*Ny–*1 // *Ny–*1 – число столбцов матрицы, задающей

внутренние узлы сетки

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*)) (2)

enddo

enddo

enddo

Если исходное уравнение имеет вид

*f*(*x*1,*x*2), (*x*1,*x*2)*G*, *a*>0,

то



.

Все зависимости алгоритма, задаваемого гнездом циклов (2), являются однородными и выражаются векторами зависимостей (0,1,0), (0,0,1), (1,0,0), (1,0,*–*1), (1,*–*1,0) (рис. 1). В случае метода Зейделя вектор истинных зависимостей (1,0,0) отсутствует.

*l*

*j*

*i*

Рисунок 1.1 – Схематичное изображение зависимостей алгоритма (2)

Отметим одну особенность шаблонных вычислений. Итерационные алгоритмы (параметр внешнего цикла является номером итерации) требуют после некоторого количества слоев-итераций (возможно, после каждой итерации) оценить погрешность, связанную со сходимостью процесса. Если внешний цикл задает временной слой, то после некоторого количества слоев (возможно, после каждого) может потребоваться пересчет временного шага. Некоторое внимание этому аспекту процесса вычислений будет далее уделено.

Отметим еще одну особенность шаблонных вычислений. В приведенных последовательных алгоритмах при вычислении нового значения *u*(*i*,*j*) используются или значения, полученные на предыдущей итерации (алгоритмы типа Якоби), или вполне конкретные значения, полученные как на предыдущей, так и на текущей итерации (алгоритмы зейделевского типа). При параллельных реализациях возможно использование значений, полученных на предыдущей и на текущей итерации, но не обязательно указанных на какой из них. Такого типа параллельные сеточные алгоритмы называются алгоритмами хаотической релаксации (chaotic relaxation) [1,2].

## 1.2 Тайлинг. Тайлинг с перекрытием вычислений

Тайлинг (выделение макроопераций для получения алгоритмов блочного типа) применяется для построения эффективных параллельных алгоритмов и для уменьшения накладных расходов на использование иерархической памяти. Основные понятия тайлинга будем предполагать известными [3–6]. В этом разделе для случая двумерного гнезда циклов проиллюстрируем технику тайлинга и введем понятие тайлинга с перекрытиями.

Рассмотрим двумерный цикл следующего вида:

do *i =* 1*, M*

do *j =* 1*, N*

\\\ Вычисления итерации (*i, j*)

enddo

enddo

Если множество итераций {(*i, j*)| 1≤*i*≤*M*, 1≤*j*≤*N*} разбить на некоторые блоки, то множество операций также разобьется на блоки-макрооперации. Процесс разбиения называется тайлингом, а блоки итераций и макрооперации называют тайлами. Опишем процесс разбиение на блоки (тайлы).

Пусть размер блока итераций равен *r*1×*r*2, 1≤*r*1≤*M*, 1≤*r*2≤*N*. Обозначим *Q*1, *Q*2 и преобразуем каждый цикл в двухуровневую циклическую конструкцию:

do *igl=* 0, *Q*1–1

do *i =* 1 + *iglr*1, min((*i gl*+1)*r*1, *M*)

do *jgl=* 0, *Q*2–1

do *j =* 1 + *jglr*2, min((*jgl*+1)*r*2, *N*)

\\ Вычисления итерации (*i,j*)

enddo

enddo

enddo

enddo

Выделим тайлы вычислений:

do *igl=* 0, *Q*1–1

do *jgl=* 0, *Q*2–1

Tile(*igl,jgl*)

enddo

enddo

Один тайл вычислений Tile(*igl,jgl*) составляют вычисления

do *i =* 1 + *iglr*1, min((*i gl*+1)*r*1, *M*)

do *j =* 1 + *jglr*2, min((*jgl*+1)*r*2, *N*)

\\ Вычисления итерации (*i,j*)

enddo

enddo

Тайлы Tile(*igl,jgl*), для которых выполняется хотя бы одно из равенств *igl=*0, *jgl=*0, *igl=Q*1–1, *jgl=Q*2–1, называются граничными.

Рассмотрим теперь тайлинг с перекрытием вычислений (overlapped tiling) [7, 8]. Тайлинг с перекрытием (тайлинг с дублированием некоторых вычислений), организованный на одном или нескольких итерационных (или временных) слоях, позволяет все вычисления расширенных тайлов производить без обращения к результатам вычислений других тайлов этих слоев.

Пусть производятся вычисления в ячейках (*i,j*), 1≤*i*≤*M*, 1≤*j*≤*N*. Введем в рассмотрение блоки вычислений, имеющие перекрытие. Ячейки (*i,j*), 1+*iglr*1≤*i*≤min((*igl*+1)*r*1, *M*), 1+*jglr*2≤*j*≤min((*jgl*+1)*r*2, *N*) каждого блока (тайла) вычислений Tile(*igl,jgl*) назовем основными. К основным ячейкам добавим смежные ячейки соседних блоков: (*iglr*1*,j*), (1+(*igl*+1)*r*1*,j*), где *jglr*2≤*j*≤1+(*jgl*+1)*r*2; (*i,jglr*2), (*i,*1+(*jgl*+1)*r*2), где *iglr*1≤*i*≤1+(*igl*+1)*r*1. Ячейки (*iglr*1*,jglr*2), (1+(*igl*+1)*r*1*, jglr*2), (1+(*igl*+1)*,jglr*2), (1+(*igl*+1)*,*1+(*jgl*+1)*r*2) назовем угловыми. Добавление смежных с какой-либо границей блока ячеек не происходит, если эта граница является границей сетки. К левой границе добавление не происходит при *igl=*0, к нижней границе добавление не происходит при *jgl=*0, к правой границе добавление не происходит при *igl=Q*1–1, к верхней границе добавление не происходит при *jgl=Q*2–1.

Таким образом, к *r*1×*r*2основнымячейкам блока добавляются так называемые гало-ячейки (halo – нимб). Если ячейки не являются границей сетки, то добавляются 2*r*1+2*r*2+4 ячеек, смежных основнымячейкам. Не граничные блоки вычислений (не граничные гало-тайлы) содержат (*r*1+2)×(*r*2+2) ячеек. Вычисления одного гало-блока (гало-тайла) с перекрытиями Tilehalo(*igl,jgl*) можно записать следующим образом:

do *i =* max(*igl r*1, 1), min(1+(*igl*+1)*r*1, *M*)

do *j =* max(*jgl r*2, 1), min(1+(*jgl*+1)*r*2, *N*)

\\\ Вычисления ячейки (*i,j*)

enddo

enddo

Границей гало-тайлов являются ячейки (*iglr*1*,j*), (1+(*igl*+1)*r*1*,j*), где *jglr*2≤*j*≤1+(*jgl*+1)*r*2, а также ячейки (*i,jglr*2), (*i,*1+(*jgl*+1)*r*2), где *iglr*1≤*i*≤1+(*igl*+1)*r*1. Назовем эту границу границей 1. Если требуется, можно гало-тайлы расширить. Назовем границей 2 гало-тайлов ячейки (*iglr*1–1*,j*), (2+(*igl*+1)*r*1*,j*), где *jglr*2–1≤*j*≤2+(*jgl*+1)*r*2, а также ячейки (*i,jglr*2–1), (*i,*2+(*jgl*+1)*r*2), где *iglr*1–1≤*i* ≤2+(*igl*+1)*r*1.

Если множество итераций имеет вид {(*i,j*)| 1≤*i*≤*M*, 1≤*j*≤*N*}, то один гало- -тайл Tilehalo(*igl,jgl*) с границей 1 и границей 2 составляют вычисления

do *i =* max(*iglr*1–1, 1), min(2+(*igl*+1)*r*1, *M*)

do *j =* max(*jglr*2–1, 1), min(2+(*jgl*+1)*r*2, *N*)

\\\ Вычисления ячейки (*i,j*)

enddo

enddo

Назовем размером перекрытия максимальное количество строк (столбцов) граничных ячеек гало-тайла, смежных каждой из четырех сторон основныхячеек. Один гало-тайл вычислений Tilehalo(*igl,jgl*) с размером перекрытия *mover* составляют вычисления

do *i =* max(*iglr*1–*mover+*1, 1), min(*mover*+(*igl*+1)*r*1, *M*)

do *j =* max(*jglr*2–*mover+*1, 1), min(*mover*+(*jgl*+1)*r*2, *N*)

\\\ Вычисления ячейки (*i,j*)

enddo

enddo

## 1.3 Блочный алгоритм, использующий скошенные циклы

Для осуществления тайлинга трехмерного (как в рассматриваемом случае) гнезда циклов надо выбрать три семейства гиперплоскостей и разбить итерационное пространство на макрооперации-тайлы. Семейства гиперплоскостей должны быть такими, чтобы макрооперации могли выполняться атомарно, как одна единица вычислений (допустимость тайлинга). Для генерации кода применяется предварительное аффинное преобразование итерационного пространства, при котором гиперплоскости, ограничивающие тайлы, становятся параллельными координатным плоскостям; далее применяется стандартная техника преобразования каждого цикла в двумерную циклическую конструкцию.

Будем использовать семейства гиперплоскостей, задаваемые ортогональными гиперплоскостям векторами (1,0,0), (1,1,0), (1,0,1). Такой набор гиперплоскостей позволяет с минимально возможными скосами циклов произвести тайлинг по всем трем координатам.

Каждый из векторов (1,0,0), (1,1,0), (1,0,1) ортогонален двум из пяти векторам зависимостей (0,1,0), (0,0,1), (1,0,0), (1,0,*–*1), (1,*–*1,0): вектор (1,0,0) ортогонален векторам зависимостей (0,1,0), (0.0.1), вектор (1,1,0) ортогонален векторам зависимостей (0,0,1), (1,*–*1,0), вектор (1,0,1), ортогонален векторам зависимостей (0,1,0), (1,0,*–*1). Теоретически считается, что чем больше плоскость «включает в себя» векторов зависимостей, тем лучше: информационный разрез, порождаемый плоскостью, не приводит к передаче информации по соответствующему направлению.

Осуществим предварительное аффинное преобразование (*l,i,j*)(*i*1,*i*2,*i*3) итерационного пространства, задаваемое равенствами

*i*1*=l, i*2*=l+i*,  *i*3*=l+j*.

Для этого выразим исходные координаты через новые координаты:

*l=i*1, *i=i*2–*i*1,  *j=i*3–*i*1.

Из системы неравенств

1*≤ i*1*≤ rit*, 1*≤ i*2–*i*1*≤ Nx–*1, 1*≤ i*3–*i*1*≤ Ny–*1,

получим области изменения параметров преобразованного алгоритма:

1*≤ i*1*≤ rit*,

*i*1*+*1 *≤ i*2 *≤* *i*1*+Nx–*1,

*i*1*+*1 *≤ i*3 *≤* *i*1*+Ny–*1.

Таким образом, в результате аффинного преобразования приходим к следующему гнезду циклов:

do *i*1*=* 1, *rit* // *i*1 – координата, порождаемая гиперплоскостями,

ортогональными вектору (1,0,0)

do *i*2*= i*1*+*1, *i*1*+ Nx–*1 // *i*2 – координата, порождаемая

гиперплоскостями, ортогональными вектору (1,1,0)

do *i*3*= i*1 *+*1, *i*1*+ Ny–*1 // *i*3 – координата, порождаемая

гиперплоскостями, ортогональными вектору (1,0,1)

*i= i*2–*i*1

*j= i*3–*i*1

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*))

enddo

enddo

enddo

Зависимости теперь задаются векторами (0,1,0), (0,0,1), (1,1,1), (1,1,0), (1,0,1) с неотрицательными координатами, поэтому тайлинг допустим.

Осуществим тайлинг – преобразуем каждый цикл в двумерную циклическую конструкцию. Каждый цикл разбивается на глобальный и локальный: глобальные циклы определяют порядок вычисления тайлов, локальные циклы определяют порядок вычисления итераций исходного алгоритма в границах одного тайла.

Пусть *r*1, *r*2, *r*3 – параметры размеров тайла (без учета границ области итераций), *Q*1, *Q*2, *Q*3 – число тайлов по измерениям. Так как число гиперплоскостей вдоль координат *i*1, *i*2, *i*3 равно соответственно *rit*, *rit+Nx–*2, *rit+Ny–*2, то *Q*1, *Q*2, *Q*3. Используется метод окаймления преобразованной области итераций: итерационная область окаймляется прямоугольным параллелепипедом, затем окаймляющая область разбивается на *Q*1×*Q*2×*Q*3 тайлов (допускаются избыточные области изменения параметров глобальных циклов). После разбиения циклов и перестановки циклов получим:

do = 0, *Q*1–1

do = 0, *Q*2–1

do *=* 0, *Q*3–1

do *i*1*=* (1+*r*1),min((+1)*r*1, *rit*) // Начало тайла Tile(,,)

do *i*2*=* max(2+*r*2, *i*1*+*1),min(1+(+1)*r*2, *i*1*+Nx*–1)

do *i*3*=* max(2+*r*3, *i*1*+*1),min(1+(+1)*r*3, *i*1*+Ny*–1)

*i= i*2–*i*1

*j= i*3–*i*1

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*))

enddo

enddo

enddo // Конец тайла Tile(,,)

enddo

enddo

enddo

Таким образом, получен зернистый (т.е. блочный) алгоритм

do = 0, *Q*1–1

do = 0, *Q*2–1

do *=* 0, *Q*3–1

Tile(,,) (3)

enddo

enddo

enddo

где Tile(,,) задается следующим образом:

do *i*1*=* (1+*r*1),min((+1)*r*1, *rit*)

do *i*2*=* max(2+*r*2, *i*1*+*1),min(1+(+1)*r*2, *i*1*+Nx*–1)

do *i*3*=* max(2+*r*3, *i*1*+*1),min(1+(+1)*r*3, *i*1*+Ny*–1)

*i= i*2–*i*1

*j= i*3–*i*1

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*)) (4)

enddo

enddo

enddo

## 1.4 Блочный алгоритм, использующий тайлинг с перекрытиями

В этом разделе применим к алгоритму (2) тайлинг с перекрытиями размера *mover*. Для этого применим обычный (без перекрытий) тайлинг по трем координатам и на каждой итерации *l* припишем тайлам примыкающие к каждой из их границ необходимое число, определяемое зависимостями алгоритма, продублированных итераций *i* или *j*.

Зависимости алгоритма (2) задаются векторами (0,1,0), (0,0,1), (1,0,0), (1,0,*–*1), (1,*–*1,0). Для осуществления обычного тайлинга приходилось производить такое предварительное аффинное преобразование, чтобы все координаты векторов зависимостей стали неотрицательными. Это приводило к скашиванию циклов. Тайлинг с перекрытиями осуществляется без скашивания циклов. Основным преимуществом получаемых зернистых алгоритмов считается возможность одновременно начинать выполнение операций многих тайлов, в то время как после скашивания требуется, как правило, разгон вычислений.

Будем требовать, чтобы каждый трехмерный гало-тайл мог выполняться независимо от других гало-тайлов, включающих операции тех же итераций *l* алгоритма (2). Операции одной итерации *l* назовем ярусом вычислений.

Для лучшего понимания рассмотрим гало-тайлы с перекрытиями размера 2 (*mover=*2). В этом случае гало-тайлы имеют три яруса вычислений. На ярусе 3 для дальнейшего использования необходимы только вычисления в основных ячейках (*i,j*), 1+*iglr*1≤*i*≤min((*igl*+1)*r*1, *Nx–*1), 1+*jglr*2≤*j*≤min((*jgl*+1)*r*2, *Ny–*1). На ярусе 2 необходимы вычисления, требуемые на ярусе 1 гало-тайла: вычисления в основных ячейках и на границах 1 гало-тайла; в угловых ячейках границы 1 вычислений не требуется. На ярусе 1 необходимы вычисления, требуемые на ярусе 2 гало-тайла. Вычисления должны осуществляться в основных ячейках и на границах 1 и 2 гало-тайла (но в пределах 1≤*i*≤*Nx–*1, 1≤*j*≤*Ny–*1); в угловых ячейках границы 2 вычислений не требуется. Для вычислений требуются три (с каждой из четырех сторон тайла) дополнительные строки или столбца массива *u*(*i*,*j*), вычисленного на предыдущей, или заданной на нулевой, итерации.

Для граничных ячеек гало-тайла используются значения *u*(*i*,*j*), полученные на предыдущей, но не на текущей, итерации. Поэтому гало-тайлы имеют дополнительные операции, определяемые векторами зависимостей (1,1,0), (1,0,1), используемыми вместо (0,1,0), (0,0,1); единичные, а не нулевые первые координаты векторов означают использование данных с предыдущей итерации. Эти дополнительные операции составляют левую и нижнюю границы гало-тайлов. Правую и верхнюю границы гало-тайлов составляют дополнительные операции, определяемые векторами зависимостей (1,*–*1,0), (1,0,*–*1).

Пусть, как и ранее, *r*1, *r*2, *r*3 – параметры размеров тайла (без перекрытий), *Q*1, *Q*2, *Q*3 – число тайлов по измерениям, *Q*1, *Q*2, *Q*3. Каждый гало-тайл выполняет вычисления в основных ячейках и на четырех границах (с левой, верхней, правой, нижней стороны). Наличие векторов зависимостей (1,1,0), (1,0,1), (1,*–*1,0), (1,0,*–*1) приводит к тому, что если используется перекрытие размера *mover*, то перекрытие такого размера должно быть с каждой из четырех сторон и должно выполняться *r*1*=mover+*1. С каждой стороны требуются *mover+*1 дополнительная строка или столбец массива *u*(*i*,*j*), вычисленного на предыдущей итерации (или заданного на нулевой итерации). Перекрытия вычислений не должны приводить к большим накладным расходам, поэтому разумно предположить, что размер перекрытия меньше размеров тайла: *mover*<*r*2, *mover*<*r*3.

Таким образом, алгоритм с перекрытиями размера *mover* (причем *mover=r*1*–*1), реализующий модифицированный (добавлены операции) алгоритм (2), можно представить следующим образом:

do *lgl*= 0, *Q*1*–*1

do *igl=* 0, *Q*2–1

do *jgl=* 0, *Q*3–1

Tilehalo(*lgl,igl,jgl*) (5)

enddo

enddo

enddo

Tilehalo(*lgl,igl,jgl*):

do *l=* (1+*lglr*1),min((*lgl*+1)*r*1, *rit*)

do *i =* max(*l*–(*lgl*+1)*r*1+*iglr*2*+*1, 1),min(–*l*+(*lgl*+1)*r*1+(*igl*+1)*r*2, *Nx*–1)

do *j =* max(*l*–(*lgl*+1)*r*1+*jglr*3*+*1, 1),min(–*l*+(*lgl*+1)*r*1+(*jgl*+1)*r*3, *Ny*–1)

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*)) (6)

enddo

enddo

enddo

# Глава 2 Базовая архитектура и принципы работы центрального процессора

Центральный процессор (ЦП; также центральное процессорное устройство - ЦПУ; [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) central processing unit, CPU, дословно  - центральное обрабатывающее устройство)  - [электронный блок](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%BE%D0%B5_%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE) либо [интегральная схема](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D1%85%D0%B5%D0%BC%D0%B0) ([микропроцессор](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80)), исполняющая [машинные инструкции](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) (код программ), главная часть [аппаратного обеспечения](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) [компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) или [программируемого логического контроллера](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D1%8B%D0%B9_%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%BB%D0%B5%D1%80). Иногда называют микропроцессором или просто процессором.

## 2.1 Кэширование

Кэширование  - это использование дополнительной быстродействующей памяти (так называемого [кэша](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88)  - [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) cache, от [фр.](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B0%D0%BD%D1%86%D1%83%D0%B7%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) cacher  - «прятать») для хранения копий блоков информации из основной (оперативной) памяти, вероятность обращения к которым в ближайшее время велика.

Различают кэши 1-, 2- и 3-го уровней (обозначаются L1, L2 и L3  - от Level 1, Level 2 и Level 3). Кэш 1-го уровня имеет наименьшую латентность (время доступа), но малый размер, кроме того, кэши первого уровня часто делаются многопортовыми. Так, процессоры AMD K8 умели производить одновременно 64-битные запись и чтение, либо два 64-битных чтения за такт, AMD K8L может производить два 128-битных чтения или записи в любой комбинации. Процессоры Intel Core 2 могут производить 128-битные запись и чтение за такт. Кэш 2-го уровня обычно имеет значительно большую латентность доступа, но его можно сделать значительно больше по размеру. Кэш 3-го уровня  - самый большой по объёму и довольно медленный, но всё же он гораздо быстрее, чем оперативная память.

Кэш микропроцессора  - [кэш](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88) (сверхоперативная память), используемый [микропроцессором](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) компьютера для уменьшения среднего времени доступа к [компьютерной памяти](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C). Является одним из верхних уровней [иерархии памяти](https://ru.wikipedia.org/wiki/%D0%98%D0%B5%D1%80%D0%B0%D1%80%D1%85%D0%B8%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8). Кэш использует небольшую, очень быструю память (обычно типа [SRAM](https://ru.wikipedia.org/wiki/SRAM_(%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C))), которая хранит копии часто используемых данных из основной памяти. Если большая часть запросов в память будет обрабатываться кэшем, средняя задержка обращения к памяти будет приближаться к задержкам работы кэша.

Когда процессору нужно обратиться в память для чтения или записи данных, он сначала проверяет, доступна ли их копия в кэше. В случае успеха проверки процессор производит операцию используя кэш, что значительно быстрее использования более медленной основной памяти. Подробнее о задержках памяти см. [Латентность](https://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D1%82%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D1%85%D0%B0%D1%80%D0%B0%D0%BA%D1%82%D0%B5%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B0_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B9_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8)) [SDRAM](https://ru.wikipedia.org/wiki/SDRAM): tCAS, tRCD, tRP, tRAS.

Большинство современных микропроцессоров для компьютеров и серверов имеют как минимум три независимых кэша: кэш инструкций для ускорения загрузки [машинного кода](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4), кэш данных для ускорения чтения и записи данных и [буфер ассоциативной трансляции](https://ru.wikipedia.org/wiki/Translation_lookaside_buffer) (TLB) для ускорения трансляции виртуальных (логических) адресов в физические, как для инструкций, так и для данных. Кэш данных часто реализуется в виде многоуровневого кэша (L1, L2, L3).

Увеличение размера кэш-памяти может положительно влиять на производительность почти всех приложений, хотя в некоторых случаях эффект незначителен. Работа кэш-памяти обычно прозрачна для программиста, однако для её эффективного использования в некоторых случаях применяются специальные алгоритмические приёмы, изменяющие порядок обхода данных в ОЗУ или повышающие их локальность (например, при [блочном умножении матриц](https://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D1%87%D0%BD%D0%BE%D0%B5_%D1%83%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86)).

## 2.2 Принцип работы кэша

Данная подглава описывает типичный кэш данных и некоторые виды кэшей инструкций; TLB может быть устроен сложнее, а кэш инструкций - проще. На диаграмме изображены основная и кэш-память.

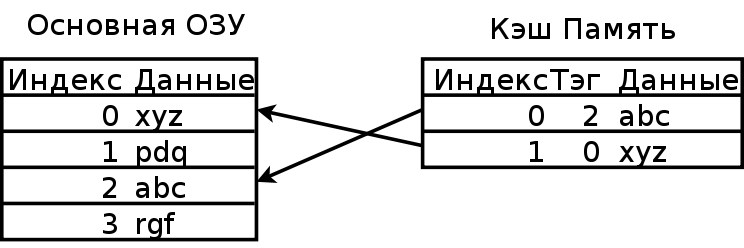


Рисунок 2.1 – Диаграмма кешей ЦПУ

Каждая строка - группа ячеек памяти содержит данные, организованные в кэш-линии. Размер каждой кэш-линии может различаться в разных процессорах, но для большинства x86-процессоров он составляет 64 байта. Размер кэш-линии обычно больше размера данных, к которому возможен доступ из одной машинной команды (типичные размеры от 1 до 16 байт). Каждая группа данных в памяти размером в 1 кэш-линию имеет порядковый номер. Для основной памяти этот номер является адресом памяти с отброшенными младшими битами. В кэше каждой кэш-линии дополнительно ставится в соответствие тег, который является адресом продублированных в этой кэш-линии данных в основной памяти.

При доступе процессора в память сначала производится проверка, хранит ли кэш запрашиваемые из памяти данные. Для этого производится сравнение адреса запроса со значениями всех тегов кэша, в которых эти данные могут храниться. Случай совпадения с тегом какой-либо кэш-линии называется попаданием в кэш ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) cache hit), обратный же случай называется кэш-промахом ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) cache miss). Попадание в кэш позволяет процессору немедленно произвести чтение или запись данных в кэш-линии с совпавшим тегом. Отношение количества попаданий в кэш к общему количеству запросов к памяти называют рейтингом попаданий ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) hit rate), оно является мерой эффективности кэша для выбранного алгоритма или программы.

В случае промаха в кэше выделяется новая запись, в тег которой записывается адрес текущего запроса, а в саму кэш-линию  - данные из памяти после их прочтения либо данные для записи в память. Промахи по чтению задерживают исполнение, поскольку они требуют запроса данных в более медленной основной памяти. Промахи по записи могут не давать задержку, поскольку записываемые данные сразу могут быть сохранены в кэше, а запись их в основную память можно произвести в фоновом режиме. Работа кэшей инструкций во многом похожа на вышеприведенный алгоритм работы кэша данных, но для инструкций выполняются только запросы на чтение. Кэши инструкций и данных могут быть разделены для увеличения производительности (принцип, используемый в [Гарвардской архитектуре](https://ru.wikipedia.org/wiki/%D0%93%D0%B0%D1%80%D0%B2%D0%B0%D1%80%D0%B4%D1%81%D0%BA%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0)) или объединены для упрощения аппаратной реализации.

Для добавления данных в кэш после кэш-промаха может потребоваться вытеснение ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) evict) ранее записанных данных. Для выбора замещаемой строки кэша используется эвристика, называемая политика замещения ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) replacement policy). Основной проблемой алгоритма является предсказание, какая строка вероятнее всего не потребуется для последующих операций. Качественные предсказания сложны, и аппаратные кэши используют простые правила, такие, как [LRU](https://ru.wikipedia.org/wiki/LRU). Пометка некоторых областей памяти как некэшируемых ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) non cacheable) улучшает производительность за счёт запрета кэширования редко используемых данных. Промахи для такой памяти не создают копию данных в кэше.

При записи данных в кэш должен существовать определенный момент времени, когда они будут записаны в основную память. Это время контролируется политикой записи ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) write policy). Для кэшей со сквозной записью ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) write-through) любая запись в кэш приводит к немедленной записи в память. Другой тип кэшей, обратная запись [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) write-back (иногда также называемый copy-back), откладывает запись на более позднее время. В таких кэшах отслеживается состояние кэш-линеек ещё не сброшенных в память (пометка битом «грязный» [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) dirty). Запись в память производится при вытеснении подобной строки из кэша. Таким образом, промах в кэше, использующем политику обратной записи, может потребовать двух операций доступа в память, один для сброса состояния старой строки и другой  - для чтения новых данных.

Существуют также смешанные политики. Кэш может быть со сквозной записью ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) write-through), но для уменьшения количества транзакций на шине записи могут временно помещаться в очередь и объединяться друг с другом.

Данные в основной памяти могут изменяться не только процессором, но и периферией, использующей [прямой доступ к памяти](https://ru.wikipedia.org/wiki/DMA), или другими процессорами в многопроцессорной системе. Изменение данных приводит к устареванию их копии в кэше (состояние stale). В другой реализации, когда один процессор изменяет данные в кэше, копии этих данных в кэшах других процессоров будут помечены как stale. Для поддержания содержимого нескольких кэшей в актуальном состоянии используется специальный протокол [кэш когерентности](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B3%D0%B5%D1%80%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%BA%D1%8D%D1%88%D0%B0).

## 2.3 Виды промахов

Промах по чтению из кэша инструкций. Обычно дает очень большую задержку, поскольку процессор не может продолжать исполнение программы (по крайней мере, текущего потока исполнения) и вынужден простаивать в ожидании загрузки инструкции из памяти.

Промах по чтению из кэша данных. Обычно дает меньшую задержку, поскольку инструкции, не зависящие от запрошенных данных, могут продолжать исполняться, пока запрос обрабатывается в основной памяти. После получения данных из памяти можно продолжать исполнение зависимых инструкций.

Промах по записи в кэш данных. Обычно дает наименьшую задержку, поскольку запись может быть поставлена в очередь и последующие инструкции практически не ограничены в своих возможностях. Процессор может продолжать свою работу, кроме случаев промаха по записи с полностью заполненной очередью.

## 2.4 Мультипроцессор и многопоточность

Мультипроцессор (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) multiprocessor, multiprocessing) - это подкласс [многопроцессорных](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) компьютерных систем, где есть несколько [процессоров](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) и одно [адресное пространство](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D1%80%D0%B5%D1%81%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE), видимое для всех процессоров. В [таксономии Флинна](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BA%D1%81%D0%BE%D0%BD%D0%BE%D0%BC%D0%B8%D1%8F_%D0%A4%D0%BB%D0%B8%D0%BD%D0%BD%D0%B0) мультипроцессоры относятся к классу [SM-MIMD](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BA%D1%81%D0%BE%D0%BD%D0%BE%D0%BC%D0%B8%D1%8F_%D0%A4%D0%BB%D0%B8%D0%BD%D0%BD%D0%B0#SM-MIMD_.28shared_memory_MIMD.29)-машин. Мультипроцессор запускает одну копию [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) с одним набором таблиц, в том числе тех, которые следят какие [страницы памяти](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) свободны.

В программировании мультипроцессоров можно использовать две [модели программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F): [многопоточность](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C), где на каждом процессоре запускается поток исполнения, и они обмениваются друг с другом данными через общие переменные в общей памяти, либо (более сложный) [message passing](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F%D0%BC%D0%B8), когда на каждом процессоре запускается отдельный процесс, и они обмениваются данными друг с другом путём обмена сообщениями. Многопоточное программирование используется либо явно (в компилируемых языках программирования с помощью систменого [API](https://ru.wikipedia.org/wiki/API) (напрмер в [C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))/[C++](https://ru.wikipedia.org/wiki/C%2B%2B) с помощью [POSIX Threads](https://ru.wikipedia.org/wiki/POSIX_Threads), а также с помощью boost::thread или std::thread в C++, начиная со стандарта C++11), в интерпретируемых языках ([Java](https://ru.wikipedia.org/wiki/Java) и [C#](https://ru.wikipedia.org/wiki/C_Sharp)) с помощью конструкций языка), либо неявно (декларативно с помощью директив компилятора ([OpenMP](https://ru.wikipedia.org/wiki/OpenMP)) или автоматически самим компилятором ([High Performance Fortran](https://ru.wikipedia.org/wiki/Fortran))).

Многопото́чность - свойство платформы (например, [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), [виртуальной машины](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0) и т. д.) или [приложения](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), состоящее в том, что [процесс](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), порождённый в операционной системе, может состоять из нескольких [потоков](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F), выполняющихся «[параллельно](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)», то есть без предписанного порядка во [времени](https://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D1%8F). При выполнении некоторых задач такое разделение может достичь более эффективного использования [ресурсов вычислительной машины](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D1%8B).

Такие потоки называют также потоками выполнения (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) [thread of execution](https://en.wikipedia.org/wiki/thread_(computer_science))); иногда называют «нитями» (буквальный перевод [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) thread) или неформально «тредами».

Сутью многопоточности является квазимногозадачность на уровне одного исполняемого процесса, то есть все потоки выполняются в [адресном пространстве](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D1%80%D0%B5%D1%81%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE) процесса. Кроме этого, все потоки процесса имеют не только общее адресное пространство, но и общие [дескрипторы файлов](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D1%8B%D0%B9_%D0%B4%D0%B5%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%80). Выполняющийся процесс имеет как минимум один (главный) поток.

Многопоточность (как доктрину [программирования](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) не следует путать ни с [многозадачностью](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C), ни с [многопроцессорностью](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C), несмотря на то, что [операционные системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), реализующие [многозадачность](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C), как правило, реализуют и многопоточность.

К достоинствам многопоточности в программировании можно отнести следующее:

* упрощение программы в некоторых случаях за счет использования общего адресного пространства;
* меньшие относительно процесса временны́е затраты на создание потока.

Повышение производительности процесса за счет распараллеливания процессорных вычислений и операций ввода-вывода.

## 2.5 Взаимодействие потоков

В многопоточной среде часто возникают проблемы, связанные с использованием параллельно исполняемыми потоками одних и тех же данных или устройств. Для решения подобных проблем используются такие методы взаимодействия потоков, как взаимоисключения (мьютексы), семафоры, критические секции и события.

* взаимоисключения (mutex, [мьютекс](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81)) - это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов (от английского mutually exclusive access - взаимно исключающий доступ) - одновременный доступ к общему ресурсу исключается. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу. Объект может поддерживать рекурсивный захват второй раз тем же потоком, увеличивая счетчик, не блокируя поток, и требуя потом многократного освобождения. Такова, например, критическая секция в [Win32](https://ru.wikipedia.org/wiki/Win32). Тем не менее, есть и такие реализации, которые не поддерживают такое и приводят к [взаимной блокировке](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) потока при попытке рекурсивного захвата. Это FAST\_MUTEX в ядре Windows;
* [семафоры](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) представляют собой доступные ресурсы, которые могут быть приобретены несколькими потоками в одно и то же время, пока пул ресурсов не опустеет. Тогда дополнительные потоки должны ждать, пока требуемое количество ресурсов не будет снова доступно. Семафоры очень эффективны, поскольку они позволяют одновременный доступ к ресурсам;
* события. Объект, хранящий в себе 1 бит информации «просигнализирован или нет», над которым определены операции «просигнализировать», «сбросить в непросигнализированное состояние» и «ожидать». Ожидание на просигнализированном событии есть отсутствие операции с немедленным продолжением исполнения потока. Ожидание на непросигнализированном событии приводит к приостановке исполнения потока до тех пор, пока другой поток (или же вторая фаза [обработчика прерывания](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) в ядре ОС) не просигнализирует событие. Возможно ожидание нескольких событий в режимах «любого» или «всех». Возможно также создание события, автоматически сбрасываемого в непросигнализированное состояние после пробуждения первого же - и единственного - ожидающего потока (такой объект используется как основа для реализации объекта «критическая секция»). Активно используются в MS Windows, как в режиме пользователя, так и в режиме ядра. Аналогичный объект имеется и в ядре Linux под названием kwait\_queue;
* [критические секции](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) обеспечивают синхронизацию подобно мьютексам, за исключением того, что объекты, представляющие критические секции, доступны в пределах одного процесса. События, мьютексы и семафоры также можно использовать в однопроцессном приложении, однако реализации критических секций в некоторых ОС (например, Windows NT) обеспечивают более быстрый и более эффективный механизм взаимно-исключающей синхронизации - операции «получить» и «освободить» на критической секции оптимизированы для случая единственного потока (отсутствия конкуренции) с целью избежать любых ведущих в ядро ОС системных вызовов. Подобно мьютексам объект, представляющий критическую секцию, может использоваться только одним потоком в данный момент времени, что делает их крайне полезными при разграничении доступа к общим ресурсам;
* [условные переменные](https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F) (condvars). Сходны с событиями, но не являются объектами, занимающими память - используется только адрес переменной, понятие «содержимое переменной» не существует, в качестве условной переменной может использоваться адрес произвольного объекта. В отличие от событий, установка условной переменной в просигнализированное состояние не влечет за собой никаких последствий в случае, если на данный момент нет потоков, ожидающих на переменной. Установка события в аналогичном случае влечет за собой запоминание состояния «просигнализировано» внутри самого события, после чего следующие потоки, желающие ожидать события, продолжают исполнение немедленно без остановки. Для полноценного использования такого объекта необходима также операция «освободить mutex и ожидать условную переменную атомарно». Активно используются в [UNIX](https://ru.wikipedia.org/wiki/UNIX)-подобных ОС. Дискуссии о преимуществах и недостатках событий и условных переменных являются заметной частью дискуссий о преимуществах и недостатках Windows и UNIX;
* порт завершения ввода-вывода (IO completion port, IOCP). Реализованный в ядре ОС и доступный через системные вызовы объект «очередь» с операциями «поместить структуру в хвост очереди» и «взять следующую структуру с головы очереди» - последний вызов приостанавливает исполнение потока в случае, если очередь пуста, и до тех пор, пока другой поток не осуществит вызов «поместить». Самой важной особенностью IOCP является то, что структуры в него могут помещаться не только явным системным вызовом из режима пользователя, но и неявно внутри ядра ОС как результат завершения асинхронной операции ввода-вывода на одном из дескрипторов файлов. Для достижения такого эффекта необходимо использовать системный вызов «связать дескриптор файла с IOCP». В этом случае помещенная в очередь структура содержит в себе код ошибки операции ввода-вывода, а также, для случая успеха этой операции - число реально введенных или выведенных байт. Реализация порта завершения также ограничивает число потоков, исполняющихся на одном процессоре/ядре после получения структуры из очереди. Объект специфичен для MS Windows, и позволяет обработку входящих запросов соединения и порций данных в серверном программном обеспечении в архитектуре, где число потоков может быть меньше числа клиентов (нет требования создавать отдельный поток с расходами ресурсов на него для каждого нового клиента);
* ERESOURCE. Мьютекс, поддерживающий рекурсивный захват, с семантикой разделяемого или эксклюзивного захвата. Семантика: объект может быть либо свободен, либо захвачен произвольным числом потоков разделяемым образом, либо захвачен всего одним потоком эксклюзивным образом. Любые попытки осуществить захваты, нарушающее это правило, приводят к блокировке потока до тех пор, пока объект не освободится так, чтобы сделать захват разрешенным. Также есть операции вида TryToAcquire  - никогда не блокирует поток, либо захватывает, либо (если нужна блокировка) возвращает FALSE, ничего не делая. Используется в ядре Windows, особенно в файловых системах - так, например, любому кем-то открытому дисковому файлу соответствует структура FCB, в которой есть 2 таких объекта для синхронизации доступа к размеру файла. Один из них - paging IO resource - захватывается эксклюзивно только в пути обрезания файла, и гарантирует, что в момент обрезания на файле нет активного ввода-вывода от кэша и от отображения в память.
* [Rundown protection](https://ru.wikipedia.org/wiki/Rundown_protection). Полудокументированный (вызовы присутствуют в файлах-заголовках, но отсутствуют в документации) объект в ядре Windows. Счетчик с операциями «увеличить», «уменьшить» и «ждать». Ожидание блокирует поток до тех пор, пока операции уменьшения не уменьшат счетчик до нуля. Кроме того, операция увеличения может отказать, и наличие активного в данный момент времени ожидания заставляет отказывать все операции увеличения.

# Глава 3 OpenMP

Существует много разных технологий параллельного программирования. Причем эти технологии отличаются не только и не столько языками программирования, сколько архитектурными подходами к построению параллельных систем.

Например, какие-то технологии предполагают построение параллельных решений на основе нескольких компьютеров (как одного, так и разных типов), другие же предполагают именно работу на одной машине с несколькими процессорными ядрами.

Системы на базе нескольких компьютеров относят к классу систем для распределенных вычислений. Подобные решения используются довольно давно. Наиболее яркий пример технологии распределенных вычислений - [MPI](http://www.viva64.com/terminology/MPI_rus.html) (Message Passing Interface - интерфейс передачи сообщений). MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные/многопроцессорные с общей/раздельной памятью), взаимного расположения ветвей (на одном процессоре или на разных).

Так как MPI предназначен для систем с раздельной памятью, то использовать его для организации параллельного процесса в системе с общей памятью не лучшая идея. Это будет слишком избыточно и сложно, поэтому-то и начали развиваться решения вроде OpenMP. Хотя ничто не мешает делать MPI-решения для одной машины.

Cистемы параллельного программирования для работы на одной машине, начали развиваться относительно недавно. Не стоит думать, что это принципиально новые идеи, но именно с приходом (вернее с предстоящим приходом) многоядерных систем на рабочий стол, программистам стоит обратить свое внимание на такие технологии как [OpenMP](http://www.viva64.com/terminology/OpenMP_rus.html), [Intel Thread Building Blocks](http://www.threadingbuildingblocks.org/), [Microsoft Parallel Extensions](http://en.wikipedia.org/wiki/Parallel_Extensions) и ряд других.

Очень важно, чтобы технология параллельного программирования поддерживала возможность делать программу параллельной постепенно. Идеальную параллельную программу надо сразу писать параллельной, а еще лучше на каком-нибудь функциональном языке, где вопрос распараллеливания вообще не стоит. Но в реальном мире вместо функционального F# есть 10 МБайт кода на C++. И этот код надо постепенно распараллеливать. В этом случае технология OpenMP (к примеру) будет очень удачным выбором. Она позволяет, выбрав в приложении наиболее нуждающиеся в параллелизации места, в первую очередь сделать параллельными именно их. На практике это выглядит так. С помощью какого-либо инструмента для профилирования программист ищет в программе «узкие места», которые работают наиболее долго. Затем эти узкие места программист делает параллельными с помощью OpenMP. После этого, можно искать следующие узкие места и так далее, до тех пор, пока не будет получена желаемая производительность приложения. Именно поэтому в частности технология OpenMP стала довольно популярной.

OpenMP (Open Multi-Processing) - это набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью (SMP-системах).

Первый стандарт OpenMP был разработан в 1997 г. как API, ориентированный на написание легко переносимых многопоточных приложений. Сначала он был основан на языке Fortran, но позднее включил в себя и языки Си и Си++.

Интерфейс OpenMP стал одной из наиболее популярных технологий параллельного программирования. OpenMP успешно используется как при программировании суперкомпьютерных систем с большим количеством процессоров, так и в настольных пользовательских системах или, например, в Xbox 360.

Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и программного обеспечения, чья работа регулируется некоммерческой организацией «OpenMP Architecture Review Board» ([ARB](http://www.openmp.org/)).

В OpenMP используется модель параллельного выполнения «ветвление-слияние». Программа OpenMP начинается как единственный поток выполнения, называемый начальным потоком. Когда поток встречает параллельную конструкцию, он создает новую группу потоков, состоящую из себя и некоторого числа дополнительных потоков, и становится главным в новой группе. Все члены новой группы (включая главный) выполняют код внутри параллельной конструкции. В конце параллельной конструкции имеется неявный барьер. После параллельной конструкции выполнение пользовательского кода продолжает только главный поток. В параллельный регион могут быть вложены другие параллельные регионы, в которых каждый поток первоначального региона становится основным для своей группы потоков. Вложенные регионы могут в свою очередь включать регионы более глубокого уровня вложенности.

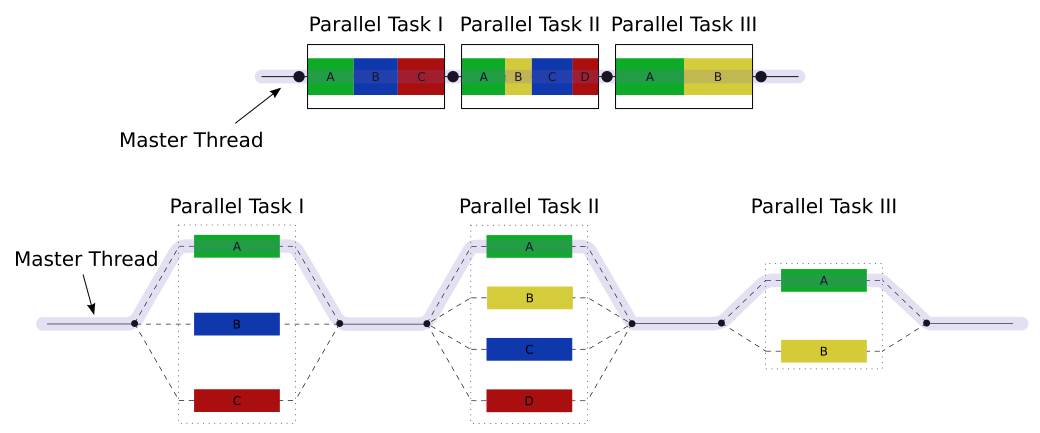


Рисунок 3.1 –Схематическое изображение модели параллельного исполнения OpenMP

Число потоков в группе, выполняющихся параллельно, можно контролировать несколькими способами. Один из них - использование переменной окружения OMP\_NUM\_THREADS. Другой способ - вызов процедуры omp\_set\_num\_threads(). Еще один способ - использование выражения num\_threads в сочетании с директивой parallel.

В [OpenMP](http://www.viva64.com/terminology/OpenMP_rus.html) существует ряд вспомогательных функций. Для их использования не забудьте подключить заголовочный файл <omp.h>.

## 3.1 Функции исполняющей среды

Эти функции позволяют запрашивать и задавать различные параметры среды OpenMP:

* omp\_get\_num\_procs - возвращает число вычислительных узлов (процессоров/ядер) в компьютере;
* omp\_in\_parallel - позволяет [потоку](http://www.viva64.com/terminology/Thread_Parallel_thread_rus.html) узнать, занимается ли он в данный момент выполнением параллельного региона;
* omp\_get\_num\_threads - возвращает число потоков, входящих в текущую группу потоков;
* omp\_set\_num\_thread - задает число потоков для выполнения следующего параллельного региона, который встретится текущему выполняемому потоку. Функция может помочь распределить ресурсы. Например, если мы одновременно обрабатываем звук и видео на процессоре с четырьмя ядрами, то можно создать для звука один поток, а для обработки видео — три;
* omp\_get\_max\_threads - возвращает максимально допустимое число нитей для использования в следующей параллельной области;
* omp\_set\_nested - разрешает или запрещает вложенный параллелизм. Если вложенный параллелизм разрешён, то каждая нить, в которой встретится описание параллельной области, породит для её выполнения новую группу нитей и станет в ней главной;
* omp\_get\_nested - возвращает, разрешен ли вложенный параллелизм или нет.

Если имя функции начинается с omp\_set\_, то ее можно вызывать только вне параллельных регионов. Все остальные функции можно использовать как внутри параллельных регионов, так и вне таковых.

## 3.2 Функции синхронизации/блокировки

OpenMP позволяет строить параллельный код без использования этих функций, так как имеются директивы, позволяющие осуществлять определенные виды синхронизации. Однако в ряде случаев эти функции удобны и даже необходимы.

В OpenMP два типа блокировок: простые и вложенные. Вложенные имеют суффикс «nest». Блокировки могут находиться в одном из трех состояний - неинициализированном, заблокированном и разблокированном.

* omp\_init\_lock/omp\_init\_nest\_lock - инициализация переменной типа omp\_lock\_t/omp\_nest\_lock\_t. Аналог InitializeCriticalSection.
* omp\_destroy\_lock/omp\_destroy\_nest\_lock - освобождение переменной типа omp\_lock\_t/omp\_nest\_lock\_t. Аналог DeleteCriticalSection.
* omp\_set\_lock/omp\_set\_nest\_lock - один поток выставляет блокировку, а остальные потоки ждут, пока поток, вызвавшая эту функцию, не снимет блокировку с помощью функции omp\_unset\_lock(). Аналог EnterCriticalSection.
* omp\_unset\_lock/omp\_unset\_nest\_lock - снятие блокировки. Аналог LeaveCriticalSection.
* omp\_test\_lock/omp\_test\_nest\_lock - неблокирующая попытка захвата замка. Данная функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1. Если замок захватить не удалось, то возвращается 0. Аналог TryEnterCriticalSection.

Простые блокировки (omp\_lock\_t) не могут быть установлены более одного раза, даже тем же потоком. Вкладываемые блокировки (omp\_nest\_lock\_t) идентичны простым с тем исключением, что когда поток пытается установить уже принадлежащую ему вкладываемую блокировку, он не блокируется.

## 3.3 Функции работы с таймерами

* omp\_get\_wtime - возвращает в вызвавшем потоке астрономическое время в секундах (вещественное число двойной точности - double), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка.

omp\_get\_wtick() - возвращает в вызвавшем потоке разрешающую способность таймера в секундах, то есть точность таймера.

if (условие) - выполнение параллельной области по условию. Создание нескольких потоков осуществляется только при выполнении некоторого условия. Если условие не выполнено, то код выполняется в последовательном режиме.

num\_threads - явное задание количества потоков, которые будут выполнять параллельную область. По умолчанию выбирается последнее значение, установленное с помощью функции omp\_set\_num\_threads().

# Глава 4 Параллельные алгоритмы для реализации на многоядерных компьютерах с общей памятью

В этой главе представлены параллельные алгоритмы численного решения двумерной задачи Дирихле для уравнения Пуассона. Алгоритмы можно использовать для OpenMP-реализаций на многоядерном CPU. Первый алгоритм, назовем его точечным параллельным алгоритмом, задает независимые вычисления в ячейках 2D расчетной сетки. Второй алгоритм, блочного типа, задает скошенный параллелизм уровня тайлов. Третий алгоритм также имеет блочный тип и использует блоки с перекрытием (блоки с избыточными вычислениями); каждый блок охватывает несколько ярусов и организован таким образом, что все вычисления блока можно производить независимо, без обращения к результатам вычислений других многоярусных блоков.

## 4.1 Точечный параллельный алгоритм

Наиболее простой параллельный алгоритм для реализации на многоядерных компьютерах с общей памятью можно получить, если считать, что на каждой итерации происходят независимые вычисления в ячейках 2D расчетной сетки:

do *l =* 1, *rit*

dopar *i =* 1, *Nx–*1

dopar *j =* 1,*Ny–*1

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*))

enddopar

enddopar

enddo

Для получения потоков (нитей) вычислений используются параллельные циклы алгоритмов. На каждой текущей итерации *l* можно вычисления в ячейке (*i,j*) выполнять независимо от вычислений в других ячейках*,* если допустить использование как обновленных, так и полученных на предыдущей итерации элементов массива *u*.

В работе [1, подраздел 11.2] рассматриваются проблемы организации потоков вычислений, связанные с оценкой погрешности такого итерационного процесса.

## 4.2 Блочный параллельный алгоритм, использующий тайлинг с перекрытиями.

Алгоритм (5), реализующий модифицированный алгоритм (2), можно для каждого параметра *lgl* внешнего цикла представить независимо выполняемыми гало-тайлами:

do *lgl*= 0, *Q*1–1

dopar *igl=* 0, *Q*2–1

dopar *jgl=* 0, *Q*3–1

Tilehalo(*lgl,igl,jgl*) (5par)

enddopar

enddopar

enddo

где Tilehalo(*lgl,igl,jgl*) имеет вид (6). Параллельные циклы могут быть использованы для организации параллельных потоков вычислений.

В частности, если в алгоритме (5) потоки задает только цикл с параметром *igl*, то получим

do *lgl=* 0, *Q*1–1

// Начало параллельной области

dopar *igl=* 0, *Q*2–1

Thread(*lgl,igl*) (9)

enddopar // Конец параллельной области

enddo

Поток Thread(*lgl,igl*) включает в себя операции гало-тайлов Tilehalo(*lgl,igl,jgl*), *jgl=*0,1,…,*Q*3–1:

do *jgl=* 0, *Q*3–1

do *l=* (1+*lglr*1),min((*lgl*+1)*r*1, *rit*)

do *i =* max(*iglr*2–*mover+*1, 1),min(*mover*+(*igl*+1)*r*2, *Nx*–1)

do *j =* max(*jglr*3–*mover+*1, 1),min(*mover*+(*jgl*+1)*r*3, *Ny*–1)

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*)) (10)

enddo

enddo

enddo

enddo

Зададим в алгоритме (9), (10) вычисления, порождаемые оценкой погрешности итерационного процесса (по аналогии с оценкой погрешности для случая точечных вычислений [1]). Каждый поток формирует локальную оценку погрешности *dthread*. После завершения вычислений поток сравнивает свою оценку *dthread* с общей оценкой погрешности *dmax*. Итерации *lgl* внешнего цикла выполняются до тех пор, пока *dmax* не станет меньше заданной величины ε или пока не будет достигнуто предельное число итераций.

do *lgl=* 0, *Q*1–1

*dmax=*0 // *dmax* – максимальное изменение значений *u* на итерации *l=*(*lgl*+1)*r*1 по сравнению с итерацией *l=*(*lgl*+1)*r*1–1

// переменная *dmax* является общей (shared) для всех потоков

dopar *igl=* 0, *Q*2–1 // Начало параллельной области

Thread(*lgl,igl*) (11)

enddopar // Конец параллельной области

if (*dmax<*ε) *lgl=Q*1–1

enddo

Поток Thread(*lgl,igl*) теперь не только включает в себя операции гало-тайлов Tilehalo(*lgl,igl,jgl*), *jgl=*0,1,…,*Q*3–1, но и формирует локальную оценку погрешности, используя две последние итерации тайлов *l=*(*lgl*+1)*r*1–1 и *l=*(*lgl*+1)*r*1:

*dthread=*0 // *dthread* – максимальное в пределах потока изменение значений *u*

на итерации *l=*(*lgl*+1)*r*1 по сравнению с итерацией *l=*(*lgl*+1)*r*1–1

// переменная *dthread* является в потоке локальной (private)

do *jgl=* 0, *Q*3–1

do *l=* (1+*lglr*1),min((*lgl*+1)*r*1, *rit*)–1

do *i =* max(*iglr*2–*mover+*1, 1),min(*mover*+(*igl*+1)*r*2, *Nx*–1)

do *j =* max(*jglr*3–*mover+*1, 1),min(*mover*+(*jgl*+1)*r*3, *Ny*–1)

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*)) (12)

enddo

enddo

enddo

// Начало последней итерации *l* в тайле

// На последней итерации *l* в тайле перекрытие вычислений не требуется,

можно положить *mover=*0

do *i =iglr*2*+*1,min((*igl*+1)*r*2, *Nx*–1)

do *j = jglr*3*+*1,min((*jgl*+1)*r*3, *Ny*–1)

*utemp=u*(*i*,*j*)

*u*(*i*,*j*)*=F*(*u*(*i*–1,*j*),*u*(*i*,*j–*1),*u*(*i*,*j*),*u*(*i*,*j+*1),*u*(*i+*1,*j*))

*d=|utemp*–*u*(*i*,*j*)|

if (*dthread<d* ) *dthread=d*

enddo

enddo

// Конец последней итерации *l* в тайле

enddo

if (*dmax<dthread*) *dmax=dthread* // сравнение и присваивание выполняются

атомарно (т.е. как одна операция)

## 4.3 О применимости рассмотренных подходов к распараллеливанию и улучшению локальности

Сформулируем требования к информационным зависимостям трехмерных (вложенности 3) гнезд циклов, при удовлетворении которых возможны рассмотренные способы организации параллельных вычислений и улучшения локальности:

* первая координата всех векторов, задающих зависимости, является положительной. Это позволяет при фиксированном значении параметра внешнего цикла выполнять итерации других циклов независимо друг от друга;
* первая координата векторов, задающих зависимости, может быть неотрицательной (т.е. не обязательно строго большей нуля), если на итерациях исходного алгоритма допускается использование как обновленных, так и не обновленных элементов массива;
* вторая и третья координаты всех векторов, задающих зависимости, являются неотрицательными (хотя бы после предварительного аффинного преобразования). Это позволяет производить разбиение пространства итераций по этим координатам (достаточные условия допустимости тайлинга) и получить (посредством задания скошенного параллелизма) блоки вычислений, которые можно выполнять независимо один от другого. Заметим, что если не использовать вычислений с перекрытиями, то рассмотренный способ организации двухуровневых вычислений допускает наличие «длинных» информационных связей (вторая и третья координаты векторов зависимостей могут принимать большие значения);
* для организации вычислений с перекрытиями знак второй и третьей координат векторов, задающих зависимости, может быть любым, но сами координаты могут принимать только небольшие, в пределах нескольких единиц, значения (иначе перекрытия заведомо слишком затратные).

## 4.4 Улучшение локальности последовательного алгоритма

В алгоритме (2) на всех вхождениях имеется пространственная локальность (предполагается, что хранение в памяти элементов массивов осуществляется по строкам). В то же время, есть следующая несогласованность использования кэш-линеек, порождаемых вхождениями *u*(*i*,*j*), *u*(*i*–1,*j*), *u*(*i+*1,*j*).

Пусть *Ny* достаточно большое. Тогда при фиксированных *i* и *j* множества элементов трех кэш-линеек, содержащих *u*(*i*,*j*), *u*(*i*–1,*j*), *u*(*i+*1,*j*), не пересекаются. Если увеличить *i* на единицу, то в двух из трех линеек содержимое уже побывало в кэше, но к моменту нового использования из кэша исчезло и требует новой загрузки из оперативной памяти. Кроме того, на каждой новой итерации *l* обновленные элементы массивов снова будут загружаться в кэш.

Универсальным способом улучшения локальности данных является тайлинг. Можно воспользоваться зернистым алгоритмом (3), (4), использующим скошенные циклы, или зернистым алгоритмом (5), (6), использующим тайлинг с перекрытиями. В этих блочных модификациях алгоритма (2) данные, загружаемые в кэш, используются на разных вхождениях согласованно.

# Глава 5 Вычислительные эксперименты

## 5.1 Задача

Рассмотрим задачу Дирихле для уравнения Пуассона:

в прямоугольной области D = {0 ≤ x ≤ 1, 0 ≤ y ≤ 2} с граничными условиями

u(0, y) = 0, u(1, y) = , u(x, 0) = x, u(x, 2) = .

Задача имеет точное решение:

Построим ее численное решение с помощью рассмотренных алгоритмов.

## 5.2 Характеристики вычислительной машины

|  |  |
| --- | --- |
| Модель | MacBook Pro (Retina, 13-inch, Mid 2014) |
| Процессор | 2.6GHz dual-core Intel Core i5 processor (Turbo Boost up to 3.1GHz) with 3MB shared L3 cache |
| Оперативная память | 16GB of 1600MHz DDR3L onboard memory |

## 5.3 Оценка погрешности реализованного алгоритма

Для оценки погрешности реализованного итерационного алгоритма проведем серию вычислительных экспериментов и сравним полученные результаты с известным точным значением в каждой точке сетки. Положим шаг *h* = 0.1 и проведем итераций вычислительного алгоритма.

Максимальное значение погрешности: 1.11022e-15

Таким образом мы показали что алгоритм сходится и дает точность близкую к машинной.

## 5.4 Оценка прироста производительности при использовании точечного параллельного алгоритма

В данной серии испытаний и последующих будем использовать следующие значения переменных:

- количество итераций

*h* = 0.0005 – шаг

*N* = 2000 – количество узлов сетки по x (размер матрицы 2001х4001)

|  |  |  |
| --- | --- | --- |
| Номер попытки | Последовательный итерационный алгоритм (сек) | Точечный параллельный алгоритм (сек) |
| 1 | 28.4757 | 11.3152 |
| 2 | 28.8058 | 11.4516 |
| 3 | 29.1063 | 11.5417 |
| 4 | 28.8365 | 11.6268 |
| 5 | 28.8900 | 11.6937 |

Таблица 5.1 – время выполнения последовательного и точечного параллельного алгоритмов

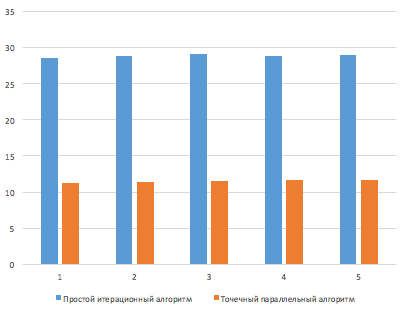


Рисунок 5.1 – время выполнения последовательного и точечного параллельного алгоритмов

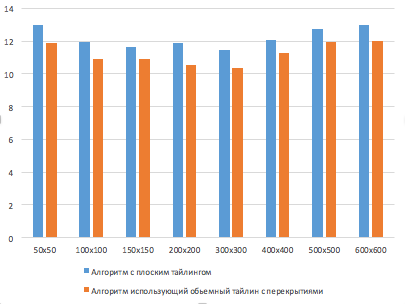
Из приведенных таблицы и графика можно сделать вывод что алгоритм хорошо поддается параллелизму, имеет хорошую локальность, дает прирост производительности соизмеримый с количеством вычислительных ядер (более чем в 2.5 раза при двух физических ядрах и технологии hyper-threading).

## 5.5 Зависимость скорости вычислений от размера тайла при плоском и трехмерном с перекрытиями алгоритмах тайлинга

В данной серии испытаний опустим зависимость скорости выполнения обьемного тайлинга от количества итераций входящих в гало тайл и положим это значение .

|  |  |  |
| --- | --- | --- |
| Размер тайла | Алгоритм с плоским тайлингом (сек) | Алгоритм использующий обьемный тайлинг с перекрытиями (сек) |
| 50x50 | 12.9508 | 11.9090 |
| 100x100 | 11.9524 | 10.9188 |
| 150x150 | 11.6168 | 10.8837 |
| 200x200 | 11.8939 | 10.5522 |
| 300x300 | 11.4300 | 10.3520 |
| 400x400 | 12.0691 | 11.2771 |
| 500x500 | 12.7543 | 11.9450 |
| 600x600 | 12.9549 | 11.9988 |

Таблица 5.2 – время выполнения алгоритмов с тайлингом в зависимости от размера тайла

Рисунок 5.2 – время выполнения алгоритмов с тайлингом в зависимости от размера тайла 

Из приведенных таблицы и графика легко заметить что алгоритм быстрее всего работает при размере тайла 300х300. Можно сделать вывод что это оптимальный размер тайла для данной модели ЦПУ.

## 5.6 Зависимость скорости вычислений от количества итераций входящих в гало тайл для алгоритма с перекрытиями

Для следующей серии испытаний положим рамер тайла 300х300 и будем рассматривать зависимость только от количества итераций входящих в гало тайл. Это оптимальный размер тайла для используемой вычислительной машины, который мы установили в предыдущей серии испытаний.

|  |  |
| --- | --- |
| Количество итераций входящих в гало тайл | Алгоритм использующий обьемный тайлинг с перекрытиями (сек) |
| 2 | 10.4585 |
| 3 | 10.6473 |
| 4 | 10.4243 |
| 5 | 10.6886 |
| 6 | 10.7193 |
| 7 | 11.6902 |
| 8 | 11.9246 |

Таблица 5.3 – время выполнения алгоритма использующего тайлинг с перекрытиями при разном количестве итераций входящих в гало тайл

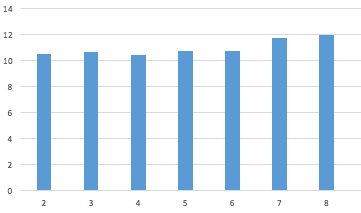


Рисунок 5.3 – время выполнения алгоритма использующего тайлинг с перекрытиями при разном количестве итераций входящих в гало тайл

В данной серии испытаний мы экспериментальным путем установили что оптимальное количество итераций входящих в гало тайл 4.

## 5.7 Сравнение производительности алгоритма использующего обьемный тайлинг с перекрытиями при оптимальном размере тайла с точечным параллельным алгоритмом

В предыдущих испытаниях было установлено что оптимальный размер тайла - 300х300, а оптимальное количство итераций входящих в гало тайл – 4. Сравним его производительность с точечным параллельным алгоритмом.

|  |  |  |
| --- | --- | --- |
| Номер попытки | Точечный параллельный алгоритм (сек) | Алгоритм использующий обьемный тайлинг с перекрытиями (сек) |
| 1 | 11.3152 | 10.4939 |
| 2 | 11.4516 | 10.4642 |
| 3 | 11.5417 | 10.5018 |
| 4 | 11.6268 | 10.3079 |
| 5 | 11.6937 | 10.3520 |

Таблица 5.4 – время выполнения алгоритма использующего тайлинг с перекрытиями в сравнении с точечным параллельным алгоритмом

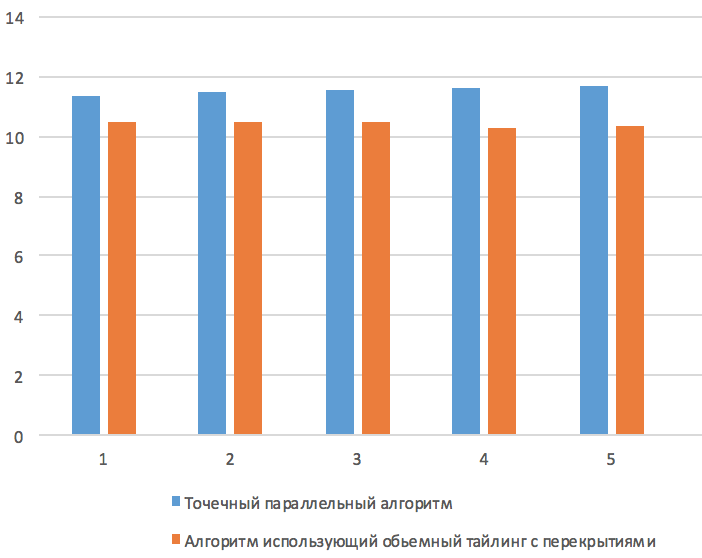


График 5.4 – время выполнения алгоритма использующего тайлинг с перекрытиями в сравнении с точечным параллельным алгоритмом

Из приведенных таблицы и графика можно заметить что и без того хорошо поддающиеся параллелизму алгоритм получает еще порядка 10% прироста производительности благодаря использованию трехмерного тайлинга с перекрытиями.

# Заключение

В ходе дипломной работы были исследованы параллельные алгоритмы решения задачи Дирихле для уравнения Пуассона и доработан алгоритм использующий трехмерный тайлинг с перекрытиями.

Была изучена структура многоядерного процессора, рассмотрены принципы работы кешей и памяти.

Были изучены современные инструменты написания программ реализующих многопоточность.

Вычислительные алгоритмы были реализованы на многоядерном процессоре с помощью технологии OpenMP, была проведена серия испытаний.

Экспериментальным путем было установлено, что итерационный метод Зейделя хорошо поддается распараллеливанию и дает прирост производительности пропорциональный количеству ядер.

Так же был экспериментальным путем установлен оптимальный размер гало тайла для алгоритма использующего обьемный тайлинг с перекрытиями и было показано что идея выделения гало тайлов, которая широко используется при расчетах на графических ускорителях, так же позволяет получить прирост производительности при реализации алгоритмов на многоядерных процессорах.

# Список использованных источников

1. ГергельВ.П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем. Учебник. М.: Изд-во Московского ун-та, 2010, 544 с. (Серия «Суперкомпьютерное образование»)
2. StrikwerdaJ.C. A convergence theorem for chaotic asynchronous relaxation // Linear algebra and its applications. 1997. Т. 253. С.15–24.
3. XueJ., CaiW. Time-minimal tiling when rise is larger than zero // Parallel Computing. 2002. Vol. 28, № 5. P. 915–939.
4. Kim D.G., Rajopadhye S. Parameterized tiling for imperfectly nested loops // Technical Report CS-09-101, Colorado State University, Department of Computer Science, February 2009. 21 p.
5. Renganarayanan L., Kim D., Rajopadhye S., Strout M.M. Parameterized tiled loops for free // ACM SIGPLAN Conference on Programming Language Design and Implementation. San Diego, California, June 2007. P. 126–138.
6. Hartono A., Baskaran M., Ramanujam J., Sadayappan P. DynTile: Parametric tiled loop generation for parallel execution on multicore processors // Proceedings of 24th International Parallel and Distributed Processing Symposium, Atlanta, April 2010. P.1–12.
7. Holewinski J., Pouchet L.-N., Sadayappan P. High-performance code generation for stencil computations on GPU architectures // Supercomputing. 2012. P. 311–320.
8. Гервич Л.Р., Штейнберг Б.Я., Юрушкин М.В. Разработка параллельных программ с оптимизацией использования структуры памяти, Ростов-на-Дону, изд-во Южного федерального университета, 2014, 120 с.
9. Renganarayanan L., Harthikote-Matha M., Dewri R., Rajopadhye S. Towards optimal multi-level tiling for stencil computations // 21st IEEE International Parallel and Distributed Processing Symposium. 2007.

# Приложение А

**Код программы реализованной для вычислительных экспериментов**

#include <iostream>

#include <fstream>

#include <omp.h>

#include <cmath>

#include <stdio.h>

#include <vector>

#include <time.h>

using namespace std;

// config

#define OMP\_THREADS\_NUM 4

// solution globals

#define Dx 1

#define Dy 2

#define N 2000

#define h ((double) Dx / N)

#define Nx ((int) (Dx / h) + 1)

#define Ny ((int) (Dy / h) + 1)

#define Rit 30

// tiling

#define r1 4

#define r2 300

#define r3 300

#define Mover 1

struct LOG\_TICK {

double time;

int i;

int j;

int threadId;

};

mutex logMutex;

vector <LOG\_TICK> LOG\_BUFFERS[OMP\_THREADS\_NUM];

double f(int i, int j) {

return 2 \* (pow(i \* h, 2) + pow(j \* h, 2));

}

double exactValue(int i, int j) {

return pow(i \* h, 2) \* pow(j \* h, 2);

}

void logSolutionError(double \*\*u) {

double maxError = u[0][0];

for (int i = 0; i < Nx; ++i) {

for (int j = 0; j < Ny; ++j) {

double err = abs(u[i][j] - exactValue(i, j));

maxError = max(maxError, err);

}

}

cout << "solution max error: " << maxError << "\n========\n";

}

void logMatrix(double \*\*u) {

cout << "U: " << "\n========\n";

for (int i = 0; i < Nx; ++i) {

for (int j = 0; j < Ny; ++j) {

cout << u[i][j] << " ";

}

cout << endl;

}

}

double\*\* allocateAndFillMatrix() {

double\*\* ary = new double\*[Nx];

for (int i = 0; i < Nx; ++i) {

ary[i] = new double[Ny];

for (int j = 0; j < Ny; ++j) {

ary[i][j] = 0;

}

}

for (int i = 0; i < Nx; ++i) {

ary[i][Ny - 1] = exactValue(i, Ny - 1);

}

for (int j = 0; j < Ny; ++j) {

ary[Nx - 1][j] = exactValue(Nx - 1, j);

}

return ary;

}

void seidel(double\*\* u, int i, int j) {

// logging

LOG\_TICK tick;

tick.i = i;

tick.j = j;

tick.threadId = omp\_get\_thread\_num();

tick.time = clock();

LOG\_BUFFERS[tick.threadId].push\_back(tick);

u[i][j] = 0.25 \* (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - pow(h, 2) \* f(i, j));

}

void solveSimple(double\*\* u) {

for (int it = 0; it < Rit; ++it) {

for (int i = 1; i < Nx - 1; ++i) {

for (int j = 1; j < Ny - 1; ++j) {

seidel(u, i, j);

}

}

}

}

void solveSimpleParallel(double\*\* u) {

#pragma omp parallel

{

for (int it = 0; it < Rit; ++it) {

// printf("it: %d(thread%d)\n", it, omp\_get\_thread\_num());

#pragma omp for

for (int i = 1; i < Nx - 1; ++i) {

for (int j = 1; j < Ny - 1; ++j) {

seidel(u, i, j);

}

}

}

}

}

void tile(double\*\* u, int ig, int jg) {

int maxI = min((ig + 1) \* r2 + 1, Nx - 1);

int maxJ = min((jg + 1) \* r3 + 1, Ny - 1);

for (int i = 1 + ig \* r2; i < maxI; ++i) {

for (int j = 1 + jg \* r3; j < maxJ; ++j) {

seidel(u, i, j);

}

}

}

void solveSimpleTiling(double\*\* u) {

int Q1 = (int) ceil(((double) Nx / r2));

int Q2 = (int) ceil(((double) Ny / r3));

#pragma omp parallel

{

for (int it = 0; it < Rit; ++it) {

#pragma omp for

for (int ig = 0; ig < Q1; ++ig) {

for (int jg = 0; jg < Q2; ++jg) {

tile(u, ig, jg);

}

}

}

}

}

void haloTile(double\*\* u, int ig, int jg, int lg, int lgr, int igr, int jgr) {

int iterationsNumber = min(lgr + 1, Rit + 1);

for (int l = 1 + lg \* r1; l < iterationsNumber; ++l) {

int maxI = min(-l + lgr + igr + 1, Nx - 1);

for (int i = max(l - lgr + ig \* r2 + 1, 1); i < maxI; ++i) {

int maxJ = min(-l + lgr + jgr + 1, Ny - 1);

for (int j = max(l - lgr + jg \* r3 + 1, 1); j < maxJ; ++j) {

seidel(u, i, j);

}

}

}

}

void solve3dTiling(double\*\* u) {

int Q1 = (int) ceil(((double) Rit / r1));

int Q2 = (int) ceil(((double) Nx / r2));

int Q3 = (int) ceil(((double) Ny / r3));

// printf("iteration tiles number: %d\n", Q1);

for (int lg = 0; lg < Q1; ++lg) {

#pragma omp parallel

{

int lgr = (lg + 1) \* r1;

#pragma omp for

for (int ig = 0; ig < Q2; ++ig) {

int igr = (ig + 1) \* r2;

for (int jg = 0; jg < Q3; ++jg) {

int jgr = (jg + 1) \* r3;

haloTile(u, ig, jg, lg, lgr, igr, jgr);

}

}

}

}

}

void showRuntime(double runtime) {

double currentTime = omp\_get\_wtime();

cout << "runtime was: " << currentTime - runtime << endl;

}

int main() {

omp\_set\_num\_threads(OMP\_THREADS\_NUM);

printf("Nx: %d, Ny: %d, h: %f, Rit: %d, (r1, r2, r3) = (%d, %d, %d)\n\_\_\_\_\_\_\_\_\_\_\n", Nx, Ny, h, Rit, r1, r2, r3);

cout << "simple:\n";

double\*\* u = allocateAndFillMatrix();

double runtime = omp\_get\_wtime();

solveSimple(u);

showRuntime(runtime);

logSolutionError(u);

cout << "simple parallel:\n";

u = allocateAndFillMatrix();

runtime = omp\_get\_wtime();

solveSimpleParallel(u);

showRuntime(runtime);

logSolutionError(u);

cout << "simple parallel tiling:\n";

u = allocateAndFillMatrix();

runtime = omp\_get\_wtime();

solveSimpleTiling(u);

showRuntime(runtime);

logSolutionError(u);

cout << "3d test parallel tiling:\n";

u = allocateAndFillMatrix();

runtime = omp\_get\_wtime();

solve3dTiling(u);

showRuntime(runtime);

logSolutionError(u);

ofstream logfile("log.txt");

for (int i = 0; i < OMP\_THREADS\_NUM; ++i) {

for (vector<LOG\_TICK>::iterator it = LOG\_BUFFERS[i].begin() ; it != LOG\_BUFFERS[i].end(); ++it) {

logfile << it->threadId << ' ' << it->i << ' ' << it->j << ' ' << it->time << endl;

}

}

logfile.close();

return 0;

}