**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**

Отчет по преддипломной практике

студента 5 курса 5 группы

«Допустить к защите» **Руководитель практики**

с предварительной оценкой Лиходед Николай Александрович

Руководитель практики профессор, доктор физико-

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ математических наук

«\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2016 г.

Минск, 2016

Оглавление

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ ШАБЛОННЫХ ВЫЧИСЛЕНИЙ 4

1.1 Итерационные процессы для решения разностной задачи Дирихле 4

1.2 Тайлинг. Тайлинг с перекрытием вычислений 7

1.3. Улучшение локальности с использованием техники аффинных преобразований и тайлинга 9

1.4. Блочный алгоритм, использующий тайлинг с перекрытиями. 10

2 Параллельные алгоритмы для реализации на многоядерных компьютерах с общей памятью 13

2.1 Точечный параллельный алгоритм. 13

2.2 Блочный параллельный алгоритм, использующий тайлинг с перекрытиями. 14

3 О применимости рассмотренных подходов к распараллеливанию и улучшению локальности 16

4. Вычислительные эксперименты 17

4.1 Задача 17

4.2 ХАРАКТЕРИСТИКИ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ 17

4.3 РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ 17

4.4 ВИЗУАЛИЗАЦИЯ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ 18

Заключение 19

Список использованных источников 20

Приложение А 21

РЕФЕРАТ

Отчет по преддипломной практике, 28 страниц, 13 рисунков, 15 формул, 6 источников.

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

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

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

**Методы работы:** 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.

# 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 этом материале 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 основного 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 задать 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

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. 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].

## 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. Улучшение локальности с использованием техники аффинных преобразований и тайлинга

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.

Предложим такие модификации алгоритма (2), в которых данные, загружаемые в кэш, будут использоваться на разных вхождениях согласованно. Для этого будем использовать тайлинг. Тайлинг является универсальным способом улучшения локальности данных.

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.

## Блочный алгоритм, использующий тайлинг с перекрытиями.

Применим к алгоритму (2) тайлинг с перекрытиями размера *mover*: применим тайлинг по трем координатам и на каждой итерации *l* припишем тайлам примыкающие к каждой из их границ продублированные *mover* итераций *i* или *j*. Зернистый алгоритм с перекрытиями можно представить следующим образом:

do *lgl*= 0, *Q*1–1

do *igl=* 0, *Q*2–1

do *jgl=* 0, *Q*3–1

// Начало тайла с перекрытиями Tilehalo(*lgl,igl,jgl*)

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*))

enddo

enddo

enddo

// Конец тайла с перекрытиями Tilehalo(*lgl,igl,jgl*)

enddo

enddo

enddo

Здесь, 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.

Зависимости алгоритма (2) задаются векторами зависимостей (0,1,0), (0,0,1), (1,0,0), (1,0,*–*1), (1,*–*1,0). Для осуществления обычного (т.е. без перекрытий) тайлинга приходилось производить такое предварительное аффинное преобразование, чтобы все координаты векторов зависимостей стали неотрицательными. Это приводило к скашиванию циклов и, как следствие, к усложнению кода.

Тайлинг с перекрытиями осуществляется без скашивания циклов, но для граничных ячеек тайла требуется использовать значения *u*(*i*,*j*), полученные на предыдущей, но не на текущей, итерации. Поэтому гало-тайлы имеют дополнительные операции, порождаемые 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, порождаемые векторами зависимостей (1,*–*1,0), (1,0,*–*1).

Отметим, что в представленном виде гало-тайлы имеют избыточные операции помимо необходимых для гало-тайлинга дополнительных операций. Рассмотрим, например, 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.

Учет избыточных вычислений усложняет запись алгоритма условиями, при которых эти вычисления не требуются. Далее будем допускать указанные не обязательные вычисления.

Таким образом, алгоритм с перекрытиями, 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(*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*)) (6)

enddo

enddo

enddo

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.

# Параллельные алгоритмы для реализации на многоядерных компьютерах с общей памятью

В этом разделе 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.

## 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.

## 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)

# 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).

# Вычислительные эксперименты

## Задача

Найти численное решение задачи 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:

## 4.2 ХАРАКТЕРИСТИКИ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ

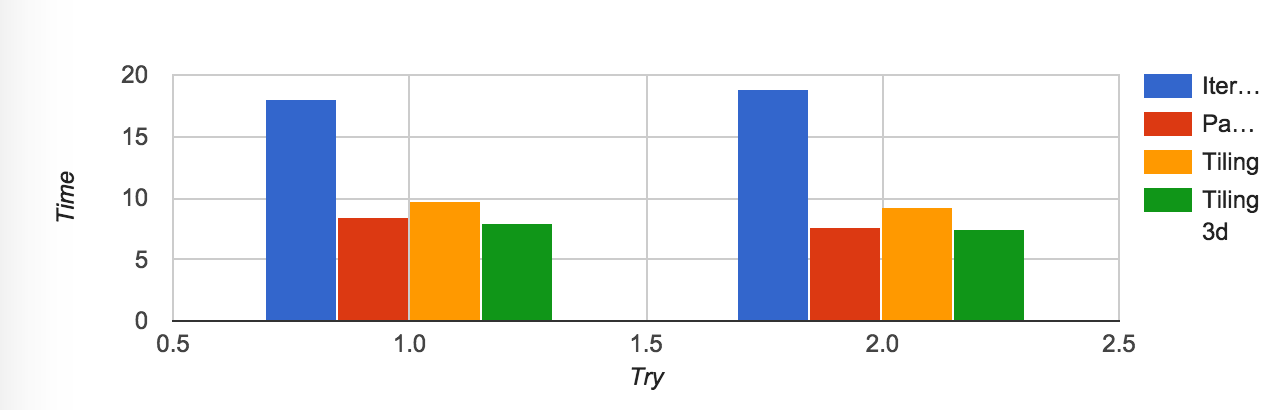
|  |  |
| --- | --- |
| Модель | MacBook Pro (Retina, 13-inch, Mid 2014) |
| Процессор | 2.6GHz dual-core Intel Core i5 processor (Turbo Boost up to 3.1GHz) with 3MB shared L3 cache |
| Оперативная память | 16GB of 1600MHz DDR3L onboard memory |

## 4.3 РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

В вычислительных экспериментах положим шаг h = 0.002, количество итераций , размеры тайла (r1, r2, r3) = (2, 100, 200) и размер перекрытий .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер попытки | Простой итерационный алгоритм | Параллельный итерационный алгоритм | Параллельный итерационный алгоритм с 2d тайлингом без перекрытий | Параллельный итерационный алгоритм с 3d тайлингом с перекрытиями |
| 1 | 17.9972 | 8.40578 | 9.6754 | 7.99101 |
| 2 | 18.9222 | 7.60442 | 9.2414 | 7.51112 |
| 3 | 18.0002 | 8.78542 | 9.4454 | 7.79101 |
| 4 | 17.5642 | 8.31565 | 9.3733 | 8.00001 |
| 5 | 16.8892 | 8.46433 | 9.5554 | 7.89121 |

## ВИЗУАЛИЗАЦИЯ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

****

# Заключение

В ходе работы были исследованы параллельные алгоритмы решения задачи Дирихле для уравнения Пуассона.

Алгоритмы были реализованы на многоядерном процессоре, была проведена серия вычислительных экспериментов.

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

Так же было установлено, что использование тайлинга с перекрытиями дает порядка 10 процентов прироста производительности.

# 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 <omp.h>

#include <cmath>

// config

#define OMP\_THREADS\_NUM 4

// solution globals

#define Dx 1

#define Dy 2

#define N 500

#define h ((double) Dx / N)

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

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

#define Rit 300

// tiling

#define r1 2

#define r2 100

#define r3 200

#define Mover 1

using namespace std;

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";

}

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) {

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) {

#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) {

for (int i = 1 + ig \* r2; i < min((ig + 1) \* r2 + 1, Nx - 1); ++i) {

for (int j = 1 + jg \* r3; j < min((jg + 1) \* r3 + 1, Ny - 1); ++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) {

for (int l = 1 + lg \* r1; l < min((lg + 1) \* r1 + 1, Rit - 1); ++l) {

for (int i = max(ig \* r2 - Mover + 1, 1); i < min(Mover + (ig + 1) \* r2 + 1, Nx - 1); ++i) {

for (int j = max(jg \* r3 - Mover + 1, 1); j < min(Mover + (jg + 1) \* r3 + 1, Ny - 1); ++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));

#pragma omp parallel

{

#pragma omp for

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

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

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

haloTile(u, ig, jg, lg);

}

}

}

}

}

void showRuntime(double runtime) {

double currentTime = omp\_get\_wtime();

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

}

int main() {

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

cout << "Nx: " << Nx << ", Ny: " << Ny << ", h: " << h << "\n\_\_\_\_\_\_\_\n";

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);

return 0;

}