УДК 004.252, 004.254, 004.272.34

Автоматизация распараллеливания программ с блочным размещением данных

Л.Р. Гервич, Е.Н. Кравченко, Б.Я. Штейнберг, М.В. Юрушкин

Южный федеральный университет, ул. Большая Садовая, 105/42, Ростов-на-Дону, 344006
E-mails: lgervith@gmail.com (Гервич Л.Р.), e.kravchenko.rnd@gmail.com (Кравченко Е.Н.), borsteinb@mail.ru (Штейнберг Б.Я.), m.yurushkin@gmail.com (Юрушкин М.В.)

Гервич Л.Р., Кравченко Е.Н., Штейнберг Б.Я., Юрушкин М.В. Автоматизация распараллеливания программ с блочным размещением данных // Сиб. журн. вычисл. математики / РАН. Сиб. отд-ние. — Новосибирск, 2015. — Т. 18, № 1. — С. 41–53.

В статье рассмотрено несколько автоматизированных приемов ускорения программ. Ускорение достигается за счет распараллеливания и оптимизации обращений к памяти. Оптимизация обращений к оперативной памяти достигается за счет перехода к блочному коду и блочным размещениям массивов. В случае распределенной памяти используются автоматизированные распределения массивов и распределения массивов с перекрытиями. Автоматизация реализуется с помощью прагм языка Си в Оптимизирующей распараллеливающей системе. Приводятся результаты численных экспериментов для задач линейной алгебры и математической физики. Некоторые демонстрационные функции этого конвертора имеют удаленный доступ.

Ключевые слова: *автоматическое распараллеливание, тайлинг, блочное распределение массивов, оптимизация обращений к памяти, размещение с перекрытиями.*

Gervich L.R., Kravchenko E.N., Steinberg B.Y., Yurushkin M.V. Automatic program parallelization with block data distribution // Siberian J. Num. Math. / Sib. Branch of Russ. Acad. of Sci. — Novosibirsk, 2015. — Vol. 18, № 1. — P. 41–53.

This paper discusses several automated methods of acceleration programs. The acceleration is achieved by parallelization and optimization of memory access. Optimization of accesses to RAM is achieved by switching to a block code and block placements arrays. When using a distributed memory, the automated distribution of arrays and array distribution with overlapping are employed. Automation is implemented using the C language with pragmas in Open Parallelizing System. This paper presents the numerical results for linear algebra and mathematical physics. Some features of this demonstration converter have a remote access to the Internet.

Key words: *automatic parallelization, tiling, memory, distributed memory, block distribution of arrays, optimization of memory, distribution with overlapping.*

Введение

Работа посвящена автоматизации блочных размещений массивов в оперативной и распределенной памяти. Такие размещения данных способствуют минимизации обращений к оперативной памяти и минимизации межпроцессорных пересылок. Разрабатываемые методики реализуются в оптимизирующей распараллеливающей системе (ОРС) Южного федерального университета [10]. Приводятся результаты численных экспериментов, которые показывают существенные преимущества описанных методов по сравнению с традиционными. Демонстрация некоторых примеров возможна в удаленном доступе [2].

Разбиение пространства итераций гнезд циклов на блоки рассматривалось во многих работах, например в [3]. Ранее, например в методе параллелепипедов [4, 6], пространство итераций разбивалось на блоки так, чтобы итерации, попадающие в блок (паралле-

лепипед) могли выполняться одновременно. Разбиение пространства итераций методом пирамид [5, 6] предназначено для распределения вычислений между процессорами без синхронизаций и пересылок данных в случае распределенной памяти. Но метод пирамид может распределять вычисления неравномерно между процессорами и недостаточно удобен для описания параллельно выполняемых процессов.

В современных процессорах обращение к оперативной памяти на порядок дольше, чем выполнение арифметических операций [7]. Для распределенной памяти эти обращения дольше арифметических операций почти на два порядка [20]. Поэтому оптимизация быстродействия программ должна быть направлена, в первую очередь, на работу с памятью.

Наилучшего быстродействия достигают программы, которые одновременно являются параллельными и учитывают иерархию памяти [1, 26]. Ускорение программ за счет параллельного выполнения или за счет использования кэш-памяти могут вступать в конфликт.

Объем кода быстрой программы может превышать объем программы простой на два порядка — это хорошо видно на задаче умножения матриц [26]. Поэтому автоматизация разработки эффективной программы может существенно сократить время разработки. Вопрос об одновременной автоматизации приемов оптимизации использования памяти и распараллеливания рассматривался в [19].

В работе рассмотрены три оптимизирующих приема для ускорения программ: блочное размещение массивов в оперативной памяти, блочно-аффинные размещения массивов [16] в распределенной памяти и размещение массивов в распределенной памяти с перекрытиями [12]. Рассматривается автоматизация этих методов на этапе компиляции и достигаемое ими ускорение.

1. Блочные размещения массивов для оптимизации использования кэш-памяти

Известно, что память вычислительных систем по степени удаленности от процессора представляет собой иерархическую структуру (основная память, кэши различного уровня, регистры). Чем ближе память к процессору, тем меньше ее размер, но выше скорость обращения к находящимся в ней данным (рис. 1).

Рис. 1. Многоуровневая память

Для того, чтобы уменьшить количество возникающих кэш-промахов, нужно изменить порядок следования операций так, чтобы над данными, находящимися в кэше, выполнялось большее количество операций [21]. Для получения такого результата исходный алгоритм часто заменяют эквивалентным ему блочным алгоритмом.

Эффект от перехода к блочным вычислениям может быть увеличен, если массивы размещать в оперативной памяти не стандартными способами (по строкам в языках Си и Паскаль или по столбцам в Фортране),

а по блокам [18, 24, 26, 30]. Это связано с тем, что данные в кэш-память поступают линейками (длиной обычно 32 или 64 байта).

Размещать массивы требуемым образом возможно на этапе выполнения либо на этапе компиляции. Размещение массивов во время выполнения влечет за собой дополнительные накладные расходы, которые могут отрицательно сказаться на производительности программы. Это особенно верно для алгоритмов, сложность которых меньше либо равна $O(N)$. Поэтому имеет смысл размещать массивы на этапе компиляции в тех случаях, когда это возможно.

В некоторых алгоритмах [23], ориентированных на оптимизацию использования иерархии памяти и, в частности кэш-памяти, используется принцип "разделяй и властвуй". Согласно этому принципу, задача разбивается на подзадачи меньшего размера (в данном случае под размером понимается количество используемых данных). Приведение алгоритма к блочному виду как раз и реализует этот принцип на практике. С результатами в этой области можно ознакомиться в [3, 9, 14, 25, 26, 28].

В статье [26] приводится высокоэффективный алгоритм блочного умножения матриц, выполняющий также и переразмещение блоков перед их умножением. Стоит отметить, что на вход этому алгоритму поступают матрицы, распределенные стандартным способом, что заставляет производить многократные блочные переразмещения на этапе выполнения алгоритма и, тем самым, увеличивает расходы по времени. Такое ограничение на способ хранения матриц сдерживает производительность библиотек численных методов (LAPACK, MKL, ATLAS и т. д.), которые обязаны соблюдать заданный стандарт.

В последнее время активно развивается высокопроизводительный пакет PLASMA, алгоритмы которого оперируют матрицами, размещенными блочно [31]. Так как компиляторы используют стандартное размещение, программист должен самостоятельно переразметить матрицы блочно. Для облегчения работы программиста в пакет PLASMA входят также вспомогательные функции, изменяющие метод размещения матриц (со стандартного на блочный и наоборот).

Таким образом, представляется актуальной задача реализации блочного размещения массивов в компиляторе. Это могло бы выполняться при наличии специальной директивы.

В компиляторе языка Си системы ОРС реализованы директивы препроцессора, которые производят блочное размещение массивов. Работа директив была протестирована на следующих программах:

1) умножение квадратных матриц;
2) возведение матрицы в квадрат;
3) LU-разложение матрицы с преобладанием элементов по главной диагонали;
4) метод Якоби решения задачи Дирихле для уравнения Лапласа.

Каждая программа была скомпилирована в своем оригинальном виде, а также с добавленными директивами. Тестирование проводилось на компьютере с процессором Intel Core i5-2410M CPU @ 2.30GHz × 4. Ниже приведены результаты сравнения производительности блочных алгоритмов, использующих стандартное размещение матриц с теми же алгоритмами, но использующими размещение по блокам (см. табл. 1).

Результаты экспериментов показали, что использование данных директив может увеличить скорость программы до 1.7 раза в зависимости от задачи. Блочное размещение замедлило скорость выполнения исходной программы решения задачи Дирихле.

Таблица 1. Результаты сравнения блочных алгоритмов, использующих распределение матриц по строкам и по блокам

Название алгоритма	Размер матриц	Стандартный алгоритм (секунды)	Блочный алгоритм			
			Размер блока	Стандартное распределение массивов (секунды)	Блочное распределение массивов (секунды)	Ускорение блочного алгоритма за счет блочного размещения массивов
Возведение матрицы в квадрат	1024	12.363	256	3.6	2.12	1.7
Умножение матриц	1024	1.463	256	1.27	1.136	1.19
LU-разложение матрицы	1024	0.82	64	0.6	0.425	1.41
Метод Якоби	4096	0.38	16	0.16	0.21	0.76

Ниже приведен блочный алгоритм решения двумерной задачи Дирихле с размещением матрицы A по строкам в виде одномерного массива. Здесь N — количество элементов на одной стороне квадратной сетки:

1) for $(di = 0;\ di < N;\ di+ = d)$
2) for $(dj = 0;\ dj < N;\ dj+ = d)$
3) for $(i = MAX(di, 1);\ i < MIN(di + d, N-1);\ i++)$
4) for $(j = MAX(dj, 1);\ j < MIN(dj + d, N-1);\ j++)$
5) $A[i*N+j] = (A[(i+1)*N+j] + A[(i-1)*N+j] + A[i*N+j+1] + A[i*N+j-1])/4$

Полученное замедление можно объяснить тем, что при вычислении каждого узла ($A[i*N+j]$) сетки происходит обращение к его соседним узлам по строке ($A[i*N+j+1]$, $A[i*N+j-1]$) и столбцу ($A[(i+1)*N+j]$, $A[(i-1)*N+j]$). Поэтому при вычислении граничных элементов одного блока будет происходить подкачка в память элементов другого блока, что увеличит количество неиспользуемых данных, попадающих в кэш.

Для поддержки автоматического блочного распределения массивов в OPC в язык Си были добавлены дополнительные директивы:

1) *#pragma ops distribute data* (имя размещаемого массива, размеры размещаемого массива, размеры блока);
2) *#pragma ops distribute revert* (имя массива).

Первая директива (*#pragma ops distribute*) используется для того, чтобы блочно переразместить массив в месте программы, где она объявлена. Для того, чтобы вернуться к стандартному размещению массива, необходимо использовать директиву *#pragma ops revert*. Ниже приведен блок кода, представляющий собой блочное умножение матриц, аннотированный описанными выше директивами, реализованными в системе OPC:

1) *#pragma ops distribute data* (*A, N, N, d, d*)
2) *#pragma ops distribute data* (*B, N, N, d, d*)
3) *#pragma ops distribute data* (*C, N, N, d, d*)

4) for $(di = 0;\ di < dcount;\ ++di)$
5) for $(dj = 0;\ dj < dcount;\ ++dj)$
6) for $(dk = 0;\ dk < dcount;\ ++dk)$
7) for $(i = 0;\ i < d;\ i++)$
8) for $(j = 0;\ j < d;\ j++)$
9) for $(k = 0;\ k < d;\ k++)$
10) $C[di*d+i][dj*d+j] = C[di*d+i][dj*d+j] + A[di*d+i][dk*d+k] * B[dk*d+k][dj*d+j]$
11) *#pragma ops distribute revert* (C)
12) *#pragma ops distribute revert* (B)
13) *#pragma ops distribute revert* (A)

Протестировать работу директив можно на других программах с помощью автоматического распараллеливателя программ с web-интерфейсом [2].

2. Блочно-аффинные размещения массивов в распределенной памяти

В оптимизирующей распараллеливающей системе (ОРС) [10] размещение данных реализовано с помощью дополнительных директив препроцессора языка Си. Данные размещаются в соответствии с некоторым блочно-аффинным размещением с перекрытиями [15, 16]. Для автоматического распараллеливания с размещением данных средствами ОРС исходная программа должна удовлетворять ряду требований. Одним из самых существенных требований является запрет на передачу распределяемых данных подпрограмме в качестве параметра.

Результатом размещения данных для архитектур с распределенной памятью является параллельная программа на языке Си с использованием библиотеки MPI [27] для организации межпроцессорных пересылок данных и синхронизации процессов. Наряду с использованием библиотеки MPI в дальнейшем предполагается использовать и другие библиотеки межпроцессорного взаимодействия, например библиотеку ShMem [32].

Работы по размещению данных также ведутся в рамках проекта DVM-системы при ИПМ им. М.В. Келдыша РАН [22]. Реализация размещения данных в ОРС отличается от реализации в DVM-системе большими возможностями операций размещения данных. Блочные размещения данных с перекрытиями в распределенной памяти для языка Fortran поддерживает система Parawise [33]. В системе Parawise параметры размещения данных указываются в диалоге с пользователем поэтапно для каждой размерности массива.

Существует много задач, в которых невозможно (из-за ограничений объема памяти) или нецелесообразно (для экономии времени выполнения) хранить все исходные данные в каждом процессорном элементе. В таких случаях возникает вопрос о разбиении данных на части и размещении этих частей в распределенной памяти. От размещения данных в распределенной памяти может существенно зависеть количество межпроцессорных пересылок и, как следствие, эффективность программы. В работах [11, 13, 17, 34] предлагаются различные способы размещения данных в распределенной памяти.

В данной работе реализованы автоматические блочно-аффинные размещения данных [15, 16].

Определение 1. Размещением m-мерного массива $X[0 \dots N_0 - 1, \dots, 0 \dots N_{m-1} - 1]$ будем называть функцию, которая каждому набору индексов $I = (I_0, \dots, I_{m-1})$ $(0 \le I_0 < N_0, \dots, 0 \le I_{m-1} < N_{m-1})$ ставит в соответствие номер модуля памяти, в котором должен находиться элемент массива $X[I_0, \dots, I_{m-1}]$ [2].

Определение 2. Блочно-аффинным по модулю p размещением m-мерного массива X с перекрытиями будем называть такое размещение, при котором элемент $X[I_0 + t_0, \dots, I_{m-1}+t_{m-1}]$ для любых целых t_i, удовлетворяющих условию $-l_i \le t_i < r_i$, находится в модуле памяти с номером

$$u = (\lfloor I_0 - d_0 \rfloor s_0 + \dots + \lfloor I_{m-1} - d_{m-1} \rfloor s_{m-1} + s_m) \mod p,$$

где $p, d_0, \dots, d_{m-1}$ — натуральные числа, $l_0, \dots, l_{m-1}, r_0, \dots, r_{m-1}$ — целые неотрицательные числа и $s_0, \dots, s_m$ — целые числа. Величины $d_0, \dots, d_{m-1}$, $l_0, \dots, l_{m-1}$, $r_0, \dots, r_{m-1}, s_0, \dots, s_m$ будем называть параметрами блочно-аффинного по модулю p размещения.

Величины перекрытий определяются числами $l_0, \dots, lm - 1, r_0, \dots, r_{m-1}$.

Для автоматической генерации параллельного MPI-кода с размещением данных в OPC в язык Си были добавлены дополнительные директивы препроцессора.

Директива *#pragma ops distribute* должна быть помещена непосредственно над объявлением динамического массива. Директива указывает OPC, что помеченный ею динамический массив является распределенным.

Пример 1. Использование директивы *#pragma ops distribute*. Массивы X и Y являются распределенными.

```
#pragma ops distribute
int *X,  **Y;
```

Директива *#pragma ops ignore* должна быть помещена непосредственно над объявлением переменной. Директива указывает OPC, что изменение значений помеченной ею переменной не должно учитываться при распараллеливании.

Пример 2. Использование директивы *#pragma ops ignore*. Изменение значений переменных i и j игнорируется при распараллеливании.

```
#pragma ops ignore
int i, j;
```

Директива *#pragma ops single_ access* должна быть помещена непосредственно над оператором в программе. Директива указывает OPC, что помеченный ею оператор после распараллеливания должен выполняться только одним процессом.

Пример 3. Использование директивы *#pragma ops single_ access*. В результате автоматического распараллеливания оператор, помеченный директивой *#pragma ops single_ access*, будет выполняться только процессом с номером 0.

```
#pragma ops distribute
double *Y;
#pragma ops single_ access
{
Y = (double*)malloc(10*sizeof(double));
```

for $(i = 0; i < 10; i = i + 1)\ Y[i] = 0;$
}

Директива *#pragma ops distribute data*(<parameters>) должна быть помещена непосредственно над некоторым оператором в программе, <parameters> представляет из себя строку вида X, SR, GR, p, m, $\dim_0, \ldots, \dim_{m-1}$, $d_0, \ldots, d_{m-1}$, $s_0, \ldots, s_m$, где X — имя размещаемого массива, SR — признак необходимости рассылки данных, GR — признак необходимости сбора данных, p — число процессов, m — размерность массива X, $\dim_0, \ldots, \dim_{m-1}$ — размеры массива X по соответствующей координате, $d_0, \ldots, d_{m-1}$, $s_0, \ldots, s_m$ — параметры блочно-аффинного размещения данных по модулю p. Директива указывает ОРС, что в процессе выполнения помеченного ею оператора распределенный массив X должен быть размещен в соответствии с блочно-аффинным размещением данных по модулю p с перекрытиями, определяемыми параметрами $d_0, \ldots, d_{m-1}, s_0, \ldots, s_m$. При этом величина перекрытий определяется автоматически.

Пример 4. Использование директивы *#pragma ops distribute data*(<parameters>). В результате автоматического распараллеливания внутри оператора, помеченного директивой *#pragma ops distribute data*(<parameters>), массив X будет размещен в соответствии с блочно-аффинным размещением данных с перекрытиями.

#pragma ops distribute
int $*X$;
#pragma ops distribute data$(X, 1, 1, 2, 1, 10, 5, 1, 0)$
{
$\ldots\ X[i] \ldots$
$\ldots\ X[i+1] \ldots$
$\ldots$
}

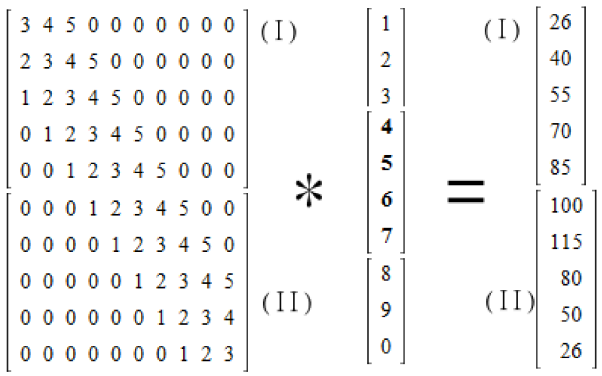
3. Размещение данных с перекрытиями

Инициализация одной пересылки данных является дорогой операцией. Поэтому для того, чтобы получить ускорение, необходимо уменьшить частоту пересылок. Это можно сделать с применением техники размещения данных с перекрытиями. Такая техника была применена в алгоритме параллельного итерационного умножения ленточной матрицы на вектор. Данный подход был описан для прямого метода решения задачи теплопроводности [12].

Рассмотрим основные идеи этого подхода на примере задачи итерационного умножения ленточной матрицы на вектор

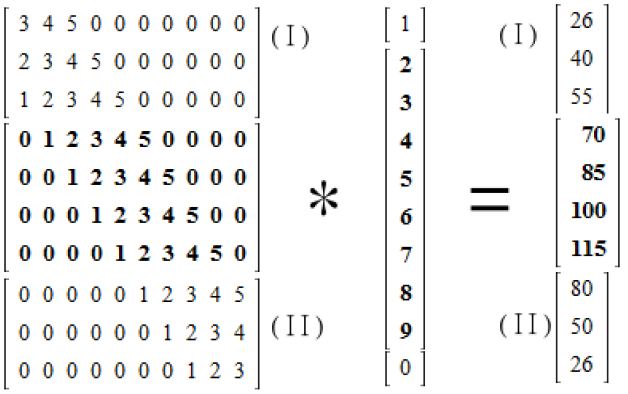
$$X^{(k+1)} = AX^{(k)}.$$

При стандартном подходе ленточная матрица A разрезается на равные полосы, вектор X разделяется на равные части, и на каждой итерации происходят пересылки элементов вектора между узлами (пример 5).



Пример 5. Одна итерация умножения $X^{(k+1)} = AX^{(k)}$ для двух вычислительных узлов. Цифрой I выделены элементы из первого узла, II — из второго, а жирным шрифтом — элементы, принадлежащие обоим узлам.

При размещении данных с перекрытием алгоритм видоизменяется: пересылки вектора происходят не на каждой итерации, а матрица разрезается так, что некоторые ее элементы могут принадлежать нескольким узлам одновременно (пример 6).



Пример 6. Одна итерация умножения $X^{(k+1)} = AX^{(k)}$ для алгоритма с перекрытием для двух вычислительных узлов. Цифрой I выделены элементы из первого узла, II — из второго, а жирным шрифтом — элементы, принадлежащие обоим узлам. Отметим, что в этом случае пересылки требуются только каждую вторую итерацию.

Размещения данных с перекрытиями могут использоваться не только для суперкомпьютеров с распределенной памятью, но и для параллельных процессоров или ускорителей, имеющих систему вычислительных устройств, каждое из которых обладает своей программируемой локальной памятью [8].

3.1. Автоматическое распараллеливание итерационного умножения ленточной матрицы на вектор

Рассмотрим задачу итерационного умножения ленточной матрицы на вектор.

Для анализа корректности и производительности автоматического распараллеливания с размещением данных в ОРС проделан ряд численных экспериментов. Одним из них является распараллеливание итерационного умножения ленточной матрицы на вектор.

Применяя директивы, описанные выше, сгенерировано 4 набора программ.

Набор 1. Исходный массив не размещен, матрица и результирующий массив размещены без перекрытий. Пересылки данных происходят на каждой итерации.

Набор 2. Исходный массив не размещен, матрица и результирующий массив размещены с перекрытиями. Пересылки данных происходят на каждой второй итерации.

Набор 3. Исходный массив, матрица и результирующий массив размещены без перекрытий. Пересылки данных происходят на каждой итерации.

Набор 4. Исходный массив, матрица и результирующий массив размещены с перекрытиями. Пересылки данных происходят на каждой второй итерации.

Полученные наборы тестов (табл. 2) прогонялись на кластере INFINI (г. Ростов-на-Дону, ЮГИНФО ЮФУ [29]). INFINI — Linux-кластер, состоящий из 20 вычислительных узлов, соединенных служебной сетью Gigabit Ethernet и скоростной коммуникационной сетью SDR Infiniband (скорость передачи 700 Мбайт/с, латентность 5 мкс). Каждый вычислительный узел представляет собой компьютер с процессором Intel Pentium 4 3.4 ГГц и оперативной памятью DDR2 2 Гбайт. Производительность каждого вычислительного узла на тесте Linpack составляет 5.8 Гфлопс, а всего кластера в целом — 115 Гфлопс.

Таблица 2. Время работы (в мкс) итерационного умножения ленточной матрицы на вектор на кластере INFINI. Число итераций — 10, размер вектора — 65536, ширина ленты — 513

Набор тестов	1 процессор	2 процессора	4 процессора	16 процессоров
Набор 1	3966889	17345575	8712618	2209653
Набор 2	3973958	14200564	7183800	1882640
Набор 3	4090970	13246628	6651545	1705230
Набор 4	4094618	13670715	6941398	1871937

3.2. Анализ количества выделяемой памяти для размещения данных с перекрытием

Рассмотрим для этой же задачи соотношение количества выделяемой памяти с перекрытием по сравнению с размещением без перекрытий.

Для алгоритма без перекрытий на каждом узле инициализируется временная матрица размера $(N/p)w$ и временный вектор размера $(N/p + 2h)$.

Для алгоритма с перекрытием — матрица размера $(N/p + 2(m-1)h)w$ и вектор размера $(N/p + 2mh)$.

Отсюда соотношение количества выделяемой памяти алгоритма с перекрытием и алгоритма без перекрытия, определяется по формуле

$$V = \frac{(N/p)\,(w+1) + 2h((m-1)\,w + m)}{(N/p)(w+1) + 2h}.$$

Тогда для вышеупомянутых размерности вектора, равной 65536, ширины ленты, равной 513, и количества узлов, равного 8:

$$V = \frac{4474368}{4203008} \approx 1.065.$$

Таким образом, расход памяти на перекрытие в данном примере составляет меньше 7%.

3.3. Автоматическое и ручное размещение данных с перекрытием для задач математической физики

Автоматическое размещение данных с перекрытием было также применено к методу Якоби решения трехмерной задачи Дирихле (табл. 3) и к сеточному методу решения уравнения теплопроводности для стержня (табл. 4).

Таблица 3. Время работы (в мкс) решения метода Якоби для задачи Дирихле на кластере INFINI. Число итераций — 10, размер сетки — $512 \times 256 \times 256$

Набор тестов	1 процессор	2 процессора	4 процессора	8 процессоров.
Набор 3	64614240	30537431	16699561	9180245
Набор 4	58120520	29634046	14995292	7761006

Таблица 4. Время работы (в мкс) решения уравнения теплопроводности на кластере INFINI. Размер сетки по времени — 1000, размер сетки по пространству — 1312072

Набор тестов	1 процессор	2 процессора	4 процессора	8 процессоров.
Набор 3	1769777	830599	427721	255050
Набор 4	58120520	1013074	544430	311646

Рассмотрим задачу Дирихле для уравнения Лапласа, заданную на декартовом произведении $D = [0,1] \times [0,1] \times [0,1]$:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0.$$

Для задачи Дирихле ставятся граничные условия первого рода.

Рассмотренный алгоритм (метод Якоби) на языке Си может быть представлен в виде

```
for (i = 0; i < max It; i++)                (итерационный процесс)
{
  for (j = 1; j < n - 1; j++)               (вычисление одной итерации метода Якоби)
  for (k = 1; k < n - 1; k++)
  for (s = 1; s < n - 1; s++)
    B[j][k][s] = 1/6 * (U[j][k-1][s] + U[j][k+1][s] + U[j-1][k][s]+
                        U[j+1][k][s] + U[j][k][s-1] + U[j][k][s+1])
  for (j = 1; j < n - 1; j++)
  for (k = 1; k < n - 1; k++)
  for (s = 1; s < n - 1; s++)
    U[j][k][s] = B[j][k][s]
}
```

Рассмотрим одномерное уравнение теплопроводности

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad 0 \le x \le 1, \quad 0 \le t \le T.$$

Алгоритм решения явной схемы одномерного уравнения теплопроводности с неоднородными граничными условиями может быть представлен в виде гнезда циклов

```
for (j = 1; j < m - 1; j++)
{
  for (i = 0; i < n - 1; i++)
    Tmp[i][j+1] = U[i][j] + τ * (U[i+1][j] - 2 * U[i][j] + U[i-1][j])/h^2
  for (i = 0; i < n - 1; i++)
    U[i][j+1] = Tmp[i][j+1]
}
```

При ручном распараллеливании с перекрытием [12] может достигаться двукратное ускорение по сравнению со стандартным распараллеливанием. В табл. 5 приведены результаты численных экспериментов для ручного распараллеливания метода Якоби для задачи Дирихле с использованием перекрытий и блочного кода.

Таблица 5. Время работы решения метода Якоби для задачи Дирихле на кластере INFINI. Число итераций — 2000, размер сетки — $500 \times 500 \times 500$, количество процессоров — 8

Время работы простого параллельного алгоритма	Время работы параллельного алгоритма с перекрытием в размещении данных и с использованием блочного кода на уровне каждого узла для оптимизации использования кэш-памяти.
26 мин 27 с	11 мин 24 с

4. Заключение

В работе приведены результаты численных экспериментов, демонстрирующие перспективные возможности автоматизированного перехода к блочному размещению массивов в оперативной и распределенной памяти.

Переход к нестандартному размещению массивов приводит к сложным индексным выражениям и, этим самым, существенно усложняет ручную разработку программы. Автоматизированное размещение массивов реализуется с помощью специальных директив компилятора — прагм. Использование прагм, с одной стороны, самостоятельный инструмент оптимизации кода, а с другой стороны, промежуточный этап на пути к автоматическому блочному распределению массивов в оптимизирующих распараллеливающих компиляторах.

Инструменты автоматизации разработки параллельных программ должны занять следующую нишу в сфере создания эффективного программного обеспечения, т. е. это программы, которые уступают рекордным несколько десятков процентов производительности, но существенно сокращают время разработки и снижают требования к квалификации программистов (и, как следствие, себестоимость).

Литература

1. **Абу-Халил Ж.М., Морылев Р.И., Штейнберг Б.Я.** Параллельный алгоритм глобального выравнивания с оптимальным использованием памяти // Современные проблемы науки и образования. — 2013. — № 1. — http://www.science-education.ru/107-8139
2. Автоматический распараллеливатель программ с web-интерфейсом. — http://ops.opsgroup.ru/opsweb-datadistr.php
3. **Арыков С.Б., Малышкин В.Э.** Система асинхронного параллельного программирования “Аспект” // Вычислительные методы и программирование. — 2008. — Т. 9, № 1. — С. 205–209.
4. **Вальковский В.А.** Параллельное выполнение циклов. Метод параллелепипедов // Кибернетика. — 1982. — № 2. — С. 51–62.
5. **Вальковский В.А.** Параллельное выполнение циклов. Метод пирамид // Кибернетика. — 1983. — № 5. — С. 51–55.
6. **Вальковский В.А.** Распараллеливание алгоритмов и программ. Структурный подход. — М.: Радио и связь, 1989.

7. **Корнеев В.В.** Проблемы программирования суперкомпьютеров на базе многоядерных мультитредовых кристаллов // Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность: Тр. Всероссийской суперкомпьютерной конференции. 21–26 сентября 2009 г., Новороссийск. — М.: Изд-во МГУ, 2009.

8. **Линев А.В., Боголюбов Д.К., Бастраков С.И.** Технологии параллельного программирования для процессоров новых архитектур. Учебник. Серия "Суперкомпьютерное образование" / Под ред. В.П. Гергеля. — М.: Изд-во Московского университета, 2010.

9. **Лиходед Н.А.** Распределение операций и массивов данных между процессорами // Программирование. — 2003. — № 3. — С. 73–80.

10. Оптимизирующая распараллеливающая система. — www.ops.rsu.ru

11. **Прангишвили И.В., Виленкин С.Я., Медведев И.Л.** Параллельные вычислительные системы с общим управлением. — М.: Энергоатомиздат, 1983.

12. **Савельев В.А.** Об оптимизации распараллеливания вычислений типа прямого метода в задаче теплопроводности для систем с распределенной памятью // Известия вузов. Северокавказский регион. Естественные науки. — 2012. — № 4. — С. 12–14.

13. **Штейнберг Б.Я.** Бесконфликтные размещения массивов при параллельных вычислениях // Кибернетика и системный анализ. — 1999. — № 1. — С. 166–178.

14. **Штейнберг Б.Я.** Блочно рекурсивное параллельное перемножение матриц // Известия вузов. Приборостроение. — 2009. — Т. 52, № 10. — С. 33–41.

15. **Штейнберг Б.Я.** Оптимизация размещения данных в параллельной памяти. — Ростов-на-Дону: Изд-во Южного федерального университета, 2010.

16. **Штейнберг Б.Я.** Блочно-аффинные размещения данных в параллельной памяти // Информационные технологии. — 2010. — № 6. — С. 36–41.

17. **Штейнберг Б.Я.** Блочно рекуррентное размещение матрицы для параллельного выполнения алгоритма Флойда // Известия вузов. Северо-кавказский регион. Естественные науки. — 2010. — № 5. — С. 31–33.

18. **Штейнберг Б.Я.** Зависимость оптимального распределения площади кристалла процессора между памятью и вычислительными ядрами от алгоритма // Тр. VI Междунар. конф. "Параллельные вычисления и задачи управления", PACO'2012. 24-26 октября 2012 г. — М.: Изд-во ИПУ им. В.А. Трапезникова РАН, 2012. — С. 99–108.

19. **Штейнберг Б.Я., Юрушкин М.В.** Новым процессорам — новые компиляторы // Открытые системы. — 2013. — № 1. — http://www.osp.ru/os/2013/01/13033990/

20. **Эйсымонт Л.К., Горбунов В.С.** На пути к экзафлопсному суперкомпьютеру: результаты, направления, тенденции // Тр. Третьего Московского суперкомпьютерного форума. — Москва, 2013. — http://www.osp.ru/docs/mscf/mscf-001.pdf

21. **Denning P.J.** The locality principle // Communications of the ACM. — 2005. — Vol. 48, iss. 7. — P. 19–24.

22. DVM system. — URL: http://www.keldysh.ru/dvm/

23. **Frigo M., Leiserson C.E., Prokop H., and Ramachandran S.** Cache-oblivious algorithms // Proc. of the 40th IEEE Symposium on Foundations of Computer Science, FOCS 99. — New York City, 1999. — P. 285–297.

24. **Herrero Josê R., Navarro Juan J.** Using non-canonical array layouts in dense matrix operations // Applied Parallel Computing. State of the Art in Scientific Computing. 8th I. Workshop, PARA 2006. — Berlin etc.: Springer-Verlag, 2007. — (Lect. Notes in Computer Science; 4699. — P. 580–588).

25. **Kulkurni D., Stumm M.** Loop and Data Transformations: A Tutorial. — Toronto: Computer Systems Research Institute, University of Toronto, 1993. — (Technical Report CSRI 337).

26. **Kazushige Goto, Robert A. van de Geijn.** Anatomy of high-performance matrix multiplication // ACM Trans. Math. Softw. — 2008. — Vol. 34, № 3. — P. 1–25.

27. Message Passing Interface (MPI) Forum Home Page. — URL: http://www.mpi-forum.org/

28. **Nikos Chrisochoides, Mokhtar Aboelaze, Elias Houstis, and Catehrine Houstis** Scalable BLAS 2 and 3 Matrix Multiplication for Sparse Banded Matrices on Distributed Memory MIMD Machines. — http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.9748

29. PARALLEL.RU. Кластерные установки России и СНГ. — URL: http://parallel.ru/russia/russian_clusters.html#infini

30. **Park N., Hong B., and Prasanna V.K.** Tiling, block data layout, and memory hierarchy performance // IEEE transactions on parallel and distributed systems. — 2003. — Vol. 14, № 7. — P. 640–654.

31. PLASMA Users' Guide. Parallel Linear Algebra Software for Multicore Architectures. Version 2.3. — USA, Knoxville: University of Tennessee, 2010.

32. SGI — HPC, Servers, Storage, Data Center Solutions, Cloud Computing. — URL: http://www.shmem.org/

33. The ParaWise Automatic Parallelization Environment. — URL: http://www.parallelsp.com/parawise.htm

34. **Wolfe M.** High Performance Compilers for Parallel Computing. — Redwood city: Addison-Wesley Publishing Company, 1996.

Поступила в редакцию 28 октября 2013 г.,
в окончательном варианте 11 марта 2014 г.