Инженерный вестник Дона, №12 (2022)
ivdon.ru/ru/magazine/archive/n12y20225/8089

Преобразования программ в Оптимизирующей распараллеливающей системе для распараллеливания на распределенную память

А.П. Баглий, Н.М. Кривошеев, Б.Я. Штейнберг, О.Б. Штейнберг

Южный федеральный университет, Ростов-на-Дону

Аннотация: В данной работе рассматривается проблема создания распараллеливающего компилятора на вычислительные системы с распределенной памятью и пути решения этой проблемы. Описаны соответствующие распараллеливающие преобразования программ, реализованные в Оптимизирующей распараллеливающей системе. Описанные преобразования автоматически определяют размещения данных в распределенной памяти с минимизацией межпроцессорных пересылок, определяют места в программе для вставки таких пересылок, приводится пример генерации параллельного кода с использованием интерфейса пересылки сообщений. Работа опирается на предыдущие публикации авторов. Разработка таких компиляторов становится актуальной для перспективных процессоров с десятками, сотнями и тысячами ядер.
Ключевые слова: автоматизация распараллеливания, распределенная память, преобразования программ, размещение данных, пересылки данных

Введение

Данная работа направлена на создание распараллеливающих компиляторов для вычислительных систем (ВС) с распределенной памятью. В статье приводятся реализованные в Оптимизирующей распараллеливающей системе (ОРС) [1] преобразования программ, которые могут использоваться для создания таких компиляторов.

Попытки создания таких компиляторов рассматривались во многих публикациях. В работе [2] отмечается, что создание компиляторов является очень сложной задачей, эффективного решения которой в данный момент не существует.

Для ВС с распределенной памятью самой длительной операцией становится межпроцессорная пересылка данных. Как показано в [3], накладные расходы, связанные с пересылками данных, приходится учитывать при отображении программ на распределенную память. Это не только затрудняет создание распараллеливающего компилятора для таких

ВС, но и существенно сужает множество эффективно распараллеливаемых программ. Однако в последнее время появляются многоядерные процессоры, иногда называемые «суперкомпьютер на кристалле», с десятками, сотнями и тысячами ядер [4, 5]. Пересылка данных между процессорными ядрами на микросхеме требует значительно меньше времени, чем на коммуникационной сети (Ethernet, Infiniband, PCI-express,…), хотя, по-прежнему остается самой длительной операцией. Это означает расширение множества эффективно распараллеливаемых программ и делает целесообразным разработку распараллеливающих компиляторов. Более того, без распараллеливающих компиляторов сдерживается развитие таких высокопроизводительных микросхем, которые могут использоваться в задачах математического моделирования [6], управления беспилотным транспортом [7] и др.

Следует отметить работы группы DVM-System [8] по генерации параллельного кода на ВС с распределенной памятью. Генерирующие автоматически параллельный код на ВС с распределенной памятью компилирующие системы DVM, Parawise и др. предполагают дописывание прагм в текст последовательной программы. Заметим, что корректное дописывание прагм возможно не к любому коду, а только к специально написанному, подготовленному. Предварительные преобразования могли бы расширить множество распараллеливаемых программ. В работе [3] рассмотрена задача трансляции высокоуровневых описаний параллельной обработки данных в программе на уровень конструкций стандарта интерфейса передачи сообщений (Message Passing Interface - MPI) для выполнения на системе с распределенной памятью.

В данной работе рассматривается распараллеливание алгоритмов, для параллельного выполнения которых достаточны блочно-аффинные размещения данных в распределенной памяти [9], а пересылки данных -

циклические или широковещательные рассылки (данные из одного процессорного элемента пересылаются в несколько или все). В [10] приводится много задач линейной алгебры и задач математической физики, для параллельного решения которых на ВС с распределенной памятью используются такие пересылки.

Некоторые оптимизации распараллеливания на распределенную память авторами уже рассматривались. В работе [11] рассмотрена задача распараллеливания программного цикла на ВС с распределенной памятью и с минимизацией межпроцессорных пересылок.

Основное отличие предлагаемого авторами подхода состоит в использовании блочно-аффинных размещений данных, которые описываются небольшим множеством параметров, что удобно для преобразования последовательной программы в параллельную для ВС с распределенной памятью.

Блочно-аффинные размещения массивов и межпроцессорные пересылки

Основная особенность параллельного выполнения цикла на ВС с распределенной памятью состоит в том, что для каждой операции ее аргументы должны быть в одном модуле распределенной памяти.

Будем полагать, что ВС состоит из p процессорных элементов (ПЭ). Каждый ПЭ состоит из процессора и собственного модуля памяти получения данных, из которого происходит быстрее, чем из модулей памяти других ПЭ. Все ПЭ занумерованы, начиная с нуля. Размещение массива в памяти – это функция, для каждого элемента массива возвращающая номер ПЭ, в котором этот элемент находится. При описании параллельных алгоритмов рассматриваются размещения матриц (двумерных массивов) «по строкам», «по столбцам», «по полосам строк», «по полосам столбцов», «по скошенным диагоналям». Эти описания, как и многие другие, могут быть описаны, как блочно-аффинные размещения по модулю количества ПЭ.

Определение 1. Пусть натуральные числа p, d_1, d_2, . . . , d_m и целые константы s_0, s_1, s_2, ..., s_m зависят только от m-мерного массива X. Блочно-аффинное по модулю p размещение m-мерного массива X – это такое размещение, при котором элемент X[i_1, i_2, ..., i_m] находится в модуле памяти ПЭ с номером:

$$u = \left(\left]\frac{i_1}{d_1}\right[* s_1 + \left]\frac{i_2}{d_2}\right[* s_2 + \cdots + \left]\frac{i_m}{d_m}\right[* s_m + s_0\right) mod\ p$$

Число s_0 показывает номер модуля памяти, в котором размещается нулевой элемент X[0,0,. . . ,0]. При описанном блочно-аффинном способе размещения m-мерный массив представляется как массив блоков размерности $d_1 * d_2 * \ldots * d_m$, который размещается так, что у каждого блока все элементы оказываются в модуле памяти одного ПЭ. Числа p, d_1, d_2, ..., d_m, s_0, s_1, s_2,..., s_m будем называть параметрами размещения.

Пример 1. Пусть задан двумерный массив X – матрица. Если d_1=d_2=1, s_0=0, s_1=1, s_2=0, то i-ая строка матрицы размещается в ПЭ с номером i mod p.

Пример 2. Пусть задан двумерный массив X – квадратная матрица размера p*p. Если d_1=d_2=1, s_0=0, s_1=1, s_2=1, то в ПЭ с номером i mod p размещается i-ая косая диагональ матрицы (диагональ, перпендикулярная главной диагонали). Получаем скошенную форму хранения матрицы [10].

Программы для ВС с распределенной памятью отличаются наличием межпроцессорных пересылок данных, которые являются самой дорогостоящей (по времени) операцией. Пересылки и должны минимизироваться при распараллеливании программ на такие архитектуры.

Пересылка – это команда коммуникационной системы.

Будем рассматривать следующую модель коммуникационной системы. Команда «пересылка» указывает, какие данные из какого ПЭ в какой следует переслать. При этом некоторые коммуникационные системы могут позволять разным ПЭ одновременно получать данные, но один ПЭ не может получать одновременно данные от нескольких ПЭ.

Если для вычислений в одном ПЭ необходимы данные, лежащие в другом ПЭ, то необходимо ожидание вычислений перед получением необходимых данных. Проще всего прервать вычисления во всех ПЭ, пока совершается межпроцессорная пересылка. Такие прерывания синхронизируют вычисления, но при большом количестве ПЭ могут создавать большие потери времени.

Можно вычисления организовать по принципу потока данных: вычисления начинаются после того, как пришли данные, т.е., приход данных инициирует вычисления.

Проверки условий распараллеливания цикла на основе анализа информационных зависимостей.

Будем рассматривать задачу параллельного выполнения цикла. Как обычно, под распараллеливанием цикла понимаем одновременное выполнение его итераций, хотя это определение предполагает уточнения, связанные с возможностями компьютера.

```
for ( j = 1 ; j < N; j = j +1) {
   Statement1(j);
   Statement2(j);
   Statement3(j) ;
}
```

Листинг №1. - Простой цикл

Для разных преобразований, включая распараллеливание, цикл должен удовлетворять разным условиям. В каждом случае, будем предполагать, что цикл удовлетворяет этим условиям, хотя иногда такие условия будем оговаривать специально. Условия корректности преобразований программ описываются в книгах по теории компиляции и в специальной литературе. В большинстве случаев цикл должен удовлетворять следующим условиям.

Счетчик цикла j не меняет значения в теле цикла; из цикла только один выход после завершения всех итераций и переход на следующую итерацию возможен только после завершения предыдущей (т.е., в теле цикла нет операторов break, continue и goto с переходом за пределы цикла);

В цикле только вхождения одномерных массивов, индексное выражение которых имеет вид (j+k), где j – счетчик цикла, k – некоторая константа или переменная, не меняющая своего значения в теле цикла.

В цикле есть только операторы присваивания.

Представленные ограничения, конечно, могут быть существенно ослаблены при их конкретной реализации.

Проверки корректности многих преобразований программ опираются на анализ информационных зависимостей. Напомним понятия информационной зависимости теории оптимизирующих/распараллеливающих преобразований программ [12, 13]. Всякому вхождению при конкретном значении индексного выражения соответствует обращение к некоторой ячейке памяти. Если при этом обращении происходит запись в ячейку памяти (вхождение в левую часть оператора присваивания, не входящее в индексное выражение другого вхождения), то такое вхождение называется генератором. Остальные вхождения называются использованиями.

Говорят, что два вхождения порождают информационную зависимость, если при некоторых допустимых значениях индексных выражений они обращаются к одной и той же ячейке памяти.

Граф информационных связей – это ориентированный граф, вершины которого соответствуют вхождениям, а дуга соединяет пару вершин (v,u), если выполняется одно из следующих условий:

1. Эти вхождения обращаются к одной и той же ячейке памяти (т.е. порождают информационную зависимость), причем вхождение v раньше, чем u и хотя бы одно из этих вхождений является генератором.

2. Вхождение u является генератором, а вхождение v принадлежит этому же оператору присваивания. Такие дуги будем называть тривиальными.

Генератор будем обозначать – out (output), а использование - in (input). Дуги графа информационных связей бывают трех типов в зависимости от типов инцидентных им вершин (см., например, [12, 13]): out-in - истинная информационная зависимость (true dependence), in-out - антизависимость (antidependence), out-out - выходная зависимость (output dependence). Если информационная зависимость связывает два использования, то мы будем говорить об in-in зависимости. Иногда будем рассматривать обобщенный граф информационных связей. Этот граф является смешанным (содержит и дуги и ребра) и представляет собой обычный граф информационных связей, к которому добавлены ребра, соединяющие вершины, соответствующие вхождениям, связанным in-in зависимостью.

Иногда бывает трудно определить, существует или нет информационная зависимость между парой вхождений. При преобразованиях программ в таких случаях предполагают худшее – считают, что такая зависимость существует, и на графе соответствующие вершины соединяют дугой.

Информационная зависимость между вхождениями называется циклически независимой (loop independent dependence), если эти вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла. Иначе зависимость называется циклически порожденной (loop carried dependence) (см., например, [12, 13]).

Вычислительные системы с распределенной памятью могут относиться к любому из двух классов параллельных вычислительных систем с одиночным потоком команд, множественным потоком данных (single instruction, multiple data – SIMD) и множественным потоком команд, множественным потоком данных (multiple instruction, multiple data - MIMD).

В [14] описаны условия, которым должны удовлетворять информационные зависимости, для распараллеливания программного цикла. Для модели выполнения MIMD не должно быть дуг циклически порожденной зависимости, для модели выполнения SIMD не должно быть дуг графа информационных связей «снизу-вверх». Следует отметить, что в [14] предполагалось, что память ВС общая. Более того, входные зависимости не принимались во внимание.

Заметим, что дуга графа информационных связей, направленная «снизу-вверх», всегда соответствует циклически порожденной зависимости.

Параллельное выполнение цикла на архитектуре SIMD, в отличие от MIMD, допускает циклически порожденные зависимости, которые направлены «сверху вниз», но на архитектуре SIMD есть проблемы с выполнением циклов, содержащих условные операторы.

Для ВС с распределенной памятью условия распараллеливания [14] сохраняют актуальность, но добавляется влияние входных зависимостей. Входные зависимости приводят к необходимости создавать межпроцессорные пересылки данных. Минимизация циклических пересылок данных рассматривалась в [11]. Следует отметить, что методы минимизации

пересылок [11] в сочетании с другими преобразованиями программ (например, с гнездованием цикла) могут дать более хорошую оптимизацию.

Многие преобразования программ, которые используются для оптимизации на компьютеры с общей памятью, могут быть полезны и для оптимизации на распределенную память. Некоторые такие преобразования реализованы в ОРС:

- Гнездование цикла.
- Перестановка циклов (заголовков циклов).
- Разбиение цикла.
- Слияние циклов.
- Раскрутка гнезда циклов.
- Инлайнинг.
- Упрощение выражений (в ОРС «Линеаризация выражений»).
- Подстановка переменных и протягивание констант.
- Поиск и вынос вычислений общих подвыражений.

Для проверки корректности преобразований в ОРС строятся граф информационных связей и управляющий граф программы.

Из внутреннего представления ОРС можно получить код программы на языке С, что позволяет, в дальнейшем, использовать такие компиляторы (для процессоров с общей памятью) с закрытым кодом, как компилятор Intel.

Граф «операторы-переменные»

В работе [11] описано построение по рассматриваемому программному циклу специального двудольного графа «операторы-переменные» (сокращенно ГОП).

Пример 3. Рассмотрим программный цикл:

```
for(int i = 1; i < N; i = i +1) {
  b[i-1] = a[i+1]+a[i+3];
  c[i+1] =b[i];
}
```

Листинг №2. – цикл для оптимизации пересылок

Такому циклу соответствует ГОП, представленный на следующем рисунке:

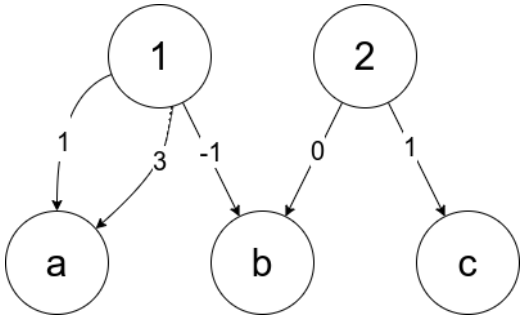

Рис. 1. – Граф «операторы-переменные» для цикла из листинга №2

Этот граф используется для поиска минимального множества межпроцессорных пересылок, необходимых для параллельного выполнения программы на ВС с распределенной памятью.

В ОРС реализован граф ГОП и его визуализация, см. рис. 2.

Любой цикл графа ГОП можно разорвать межпроцессорной пересылкой, которая вставляется в соответствующий программный цикл. Поэтому достаточно выбрать любой базис пространства циклических

векторов и каждый вектор разорвать. Результат поиска минимального числа пересылок показан на рис. 3.

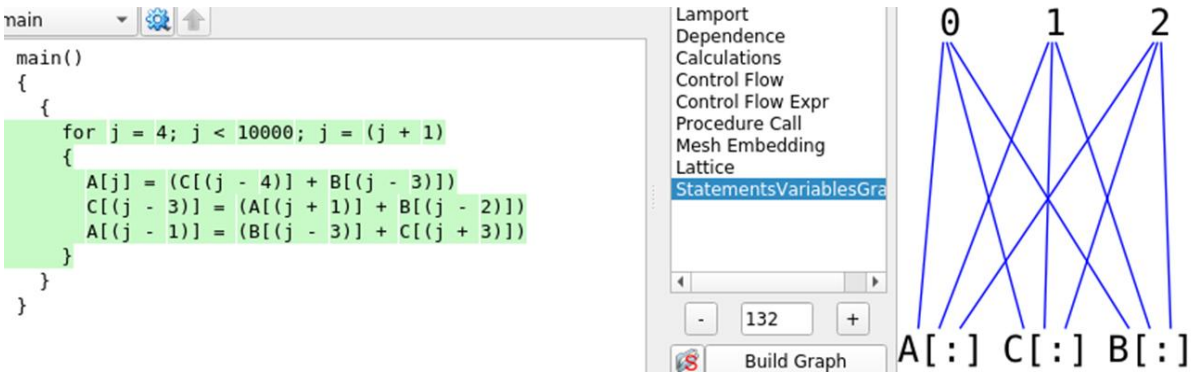

Рис. 2. - Построение и визуализация ГОП в OPC

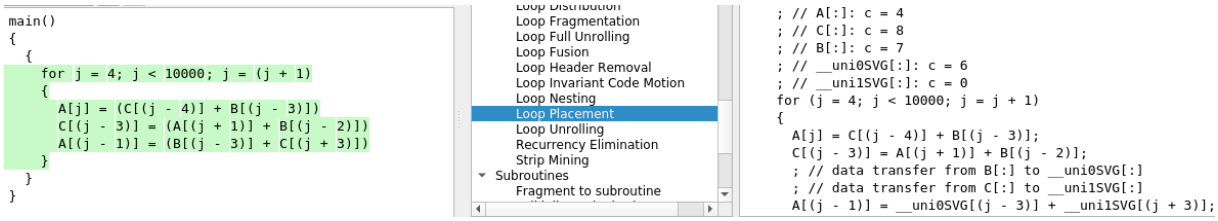

Рис. 3. – Снимок экрана OPC, демонстрирующий определение необходимых пересылок данных и их мест в программе (в код программы вставлены комментарии, которые должны заменяться функциями, выполняющими пересылки).

Распараллеливание рекуррентных программных циклов

Будем рассматривать обобщенный рекуррентный программный цикл следующего вида

```
For (j = 1; j < = N; j = j+1) {
X[j] = Gi (X[j-1]) }
```

```
}
```

Здесь X[j] – одномерный массив, G_i – функция, удовлетворяющая условиям, описанным в [17]. Примерами таких функций являются, например, аффинные и дробно-линейные функции.

Пример 4. Функция G_i описывается элементами массивов A[i], B[i]

```
Function G(X,i) //аффинная рекуррентная зависимость
Return A[i]*X+B[i]
```

Пример 5. Функция G_i описывается элементами массивов A[i], B[i], C[i], D[i].

```
Function G(X,i) // дробно-линейная рекуррентная зависимость
Return (A[i]*X+B[i])/ (C[i]*X+D[i])
```

Таблица №1

Размещение функций G_i для p=4

ПЭ0	ПЭ1	ПЭ2	ПЭ3
G_1	$G_{N/p+1}$	$G_{2*N/p+1}$	$G_{3*N/p+1}$
G_2	$G_{N/p+2}$	$G_{2*N/p+2}$	$G_{3*N/p+2}$
…	…	…	…
$G_{N/p}$	$G_{2*N/p}$	$G_{3*N/p}$	$G_{3*N/p}$

В [15] приводятся примеры ускорения приведенного цикла после распараллеливания на процессоре с общей памятью в несколько раз (на 32 ядрах до 6 раз). В данной работе алгоритм адаптируется к ВС с распределенной памятью.

Рассмотрим блочно-аффинное размещение функций G_i с параметрами s0 = (p-1), s1 = 1, d1 = N/p (полагаем, что N делится на p). Т.е. функция G_i находится в ПЭ с номером]i/(N/p) [-1 mod p

Рассмотрим блочно-аффинное описание размещения массива X с параметрами s0 = 0, s1 = 1, d1 = N/p.

Таблица №2

Размещение вычисляемого массива X для p=4

ПЭ0	ПЭ1	ПЭ2	ПЭ3
X_1	$X_{N/p+1}$	$X_{2*N/p+1}$	$X_{3*N/p+1}$
X2	$X_{N/p+2}$	$X_{2*N/p+2}$	$X_{3*N/p+2}$
…	…	…	…
$X_{N/p}$	$X_{2*N/p}$	$X_{3*N/p}$	$X_{3*N/p}$

Исходный цикл преобразуется к виду, в котором 2 цикла могут выполняться параллельно.

Символом # будем обозначать суперпозицию отображений. Псевдокод параллельного алгоритма:

```
Par For (k = 1; k < = N/p; k = k+1) // ФРАГМЕНТ 1
{
Fk = E //(тождественная функция)
For (j = 1; j < = N/p; j = j+1)
{//Вычисление суперпозиций отображений в каждом ПЭ
Fk = Fk # Gk*N/p+j
}
}
For (k = 1; k < = p; k = k+1) { // ФРАГМЕНТ 2
```

```
X[k*N/p] = Fk (X[(k-1)*N/p])

Par For (k = 1; k < = N/p; k = k+1) // ФРАГМЕНТ 3
{
For (j = 1; j < = N; j = j+1)
{//Вычисление суперпозиций отображений в каждом ПЭ
X[k*N/p+ j] = Gk*N/p+j (X[k*N/p+ j-1])
}
}
```

Результирующий код состоит из 3 фрагментов. Последний фрагмент параллельно вычисляет результирующие значения массива X и должен это делать в p раз быстрее последовательного варианта. Средний фрагмент – это короткий цикл с p итерациями. Первый фрагмент разберем подробнее.

Для случая аффинной рекуррентной зависимости функции F_k (и $G_{k*N/p+j}$) занимают мало памяти и помещаются в регистрах процессора. Промежуточные значения F_k не сбрасываются из регистров в оперативную память. По этой причине первый двойной цикл вычисляет суперпозиции $F_k = F_k \# G_{k*N/p+j}$ очень быстро и дает ускорение на современных массовых многоядерных процессорах.

На перспективных процессорах, у которых кэш-память позволяет не сбрасывать промежуточные данные в оперативную память, в этой кэш-памяти можно хранить больше промежуточных данных, чем в регистрах. По этой причине, эффективным может стать распараллеливание цикла с функцией Function G (X, i), в которой A[i] и B[i] – матрицы, а X – вектор.

Распараллеливание программного цикла с помощью стандарта MPI

В OPC разработано преобразование "блочно-аффинное размещение массивов". Для указания параметров размещения, перед преобразованием

используются прагмы. Ключевые фрагменты программы умножения матриц, к которой применено это преобразование с помощью OPC, показаны в листинге №3. Прагмы для указания параметров размещения имеют следующий вид:

```
#pragma ops distribute data(A,1,0,PROCESSES_COUNT,2,N,N,D_0,N,1,0,0,0,
0)
#pragma ops distribute data(C,0,1,PROCESSES_COUNT,2,N,N,D_0,N,1,0,0,0,
0)
#pragma ops parallel_loop_nesting
```

В OPC реализованы блочно-аффинные размещения массивов и автоматическая генерация некоторых функций библиотеки MPI.

Рассмотрим пример автоматического распараллеливания программы перемножения матриц (приведены ключевые фрагменты программы):

```
#pragma ops ignore
    int i, j, k;
#pragma ops distribute 0
    double **C;
#pragma ops distribute 1
    double **A;
    double **B;
#pragma ops single_access 0
    {  C = (double**)malloc(N * sizeof(double*));
        for (i = 0; i < N; i = i + 1) {
            C[i] = (double*)malloc(N * sizeof(double));  } }
#pragma ops single_access 1
    {
```

```
        A = (double**)malloc(N * sizeof(double*));

    …   // инициализация матрицы A
    }
    B = (double**)malloc(N * sizeof(double*));
    … // инициализация матрицы B
#pragma ops distribute data(A,1,0,NUMPROC,2,N,N,D_0,N,1,0, 0,0,0)
#pragma ops distribute data(C,0,1,NUMPROC,2 N,N,D_0,N,1,0,0,0,0)
    {
#pragma ops parallel_loop_nesting
        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++)  {
                C[i][j] = 0;
                for(k = 0; k < N; k++)
                {   C[i][j] = C[i][j] + A[i][k] * B[k][j]; }}}}
```

Листинг №3. – Результат применения преобразования «блочно-аффинное размещение массивов» с распараллеливанием программы умножения матриц в ОРС.

Следует отметить, что для автоматического распараллеливания на распределенную память использование библиотек для Симметричной иерархической памяти (Symmetric Hierarchical MEMory – SHMEM) может быть более эффективно, чем MPI, поскольку требует меньше тактов на организацию параллельных процессов.

Заключение

Данная статья представляет собой шаг на пути к созданию оптимизирующих распараллеливающих компиляторов на высокопроизводительные системы на кристалле нового поколения типа

«суперкомпьютер на кристалле». В статье представлены преобразования программ для вычислительной системы с распределенной памятью, которые не типичны для распараллеливающих компиляторов на вычислительные системы с общей памятью. Описана экспериментальная реализация таких преобразований в ОРС. Статья демонстрирует принципиальную возможность создания распараллеливающего компилятора на ВС с распределенной памятью и описывает путь к такой цели, содержащий реализацию описанных преобразований.

Исследование выполнено за счет гранта Российского научного фонда № 22-21-00671, https://rscf.ru/project/22-21-00671/

Литература

1. Оптимизирующая распараллеливающая система. URL: ops.rsu.ru (дата обращения 11.12.2022)

2. Bondhugula U. Automatic distributed-memory parallelization and codegeneration using the polyhedral framework, Technical report, ISc-CSA-TR-2011-3, 2011, 10 pp.

3. Kwon D., Han S., Kim H. MPI backend for an automatic parallelizing compiler // Proceedings Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN’99). — 06.1999. — С. 152—157. — DOI: 10.1109/ISPAN.1999.778932. — ISSN: 1087-4089.

4. SoC Esperanto URL: esperanto.ai/technology/ (дата обращения 26.03.2022).

5. Процессор НТЦ "Модуль" URL: cnews.ru/news/top/2019-03-06_svet_uvidel_moshchnejshij_rossijskij_nejroprotsessor (дата обращения 26.03.2022).

6. Медакин П.О., Никулин Р.Н., Авдеюк О.А., Королева И.Ю., Павлова Е.С., Лемешкина И.Г. Векторизация и распараллеливание метода «частица-частица» // Инженерный вестник Дона, 2021, №1. URL: ivdon.ru/magazine/archive/n1y2021/6800/.

7. Винокурский Д.Л., Ганьшин К.Ю., Мезенцева О.С., Самойлов Ф.В. Генерация траекторий оптимальных кривых беспилотного летательного аппарата для обхода статического препятствия // Инженерный вестник Дона, 2021, №9. URL: ivdon.ru/magazine/archive/n9y2021/7192/.

8. DVM-система разработки параллельных программ URL: dvm-system.org/ru/about/ (дата обр. 26.03.2022).

9. Штейнберг Б.Я. Оптимизация размещения данных в параллельной памяти // Ростов-на-Дону: изд-во Южного федерального университета, 2010, 255 с.

10. Прангишвили И.В., Виленкин С.Я., Медведев И.Л. Параллельные вычислительные системы с общим управлением // М.: Энергоатомиздат, 1983, 312 с.

11. Krivosheev N.M., Steinberg B.Ya. Algorithm for searching minimum inter-node data transfers. //«Procedia Computer Science», 10th International Young Scientist Conference on Computational Science, YSC 2021, 1-3 July 2021, pp. 306-313.

12. Падуа Д., Вольф М., Хаммер К. Векторизация программ: теория, методы, реализация. // Сборник переводов статей М.: Мир, 1991. С. 246 - 267.

13. Штейнберг Б.Я. Математические методы распараллеливания рекуррентных программных циклов для суперкомпьютеров с параллельной памятью // Ростов-на-Дону: изд-во Ростовского университета, 2004, 192 с.

14. Lamport L. The parallel execution of DO loops // Commun. ACM, 1974, vol. 17, №2, pp. 83-93.

15. Steinberg O.B. Parallelization of recurrent loops due to the preliminary computation of superpositions // Programming & Computer Software (Bulletin SUSU MMCS), Ser. Mathematical Modelling , 2020 , T. 13 , V. 3 , P. 59–67 , DOI: doi.org/10.14529/mmp200305

References

1. Optimiziruyushhaya rasparallelivayushhaya Sistema [Optimizing parallelizing system]. URL: ops.rsu.ru (date accessed 11.12.2022)

2. Bondhugula U. Automatic distributed-memory parallelization and codegeneration using the polyhedral framework. Technical report, ISc-CSA-TR-2011-3, 2011, 10 pp.

3. Kwon D., Han S., Kim H. MPI backend for an automatic parallelizing compiler. Proceedings Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN’99). 06.1999. S. 152—157. DOI: 10.1109/ISPAN.1999.778932. ISSN: 1087-4089.

4. System on chip Esperanto URL: esperanto.ai/technology/ (date accessed 26.03.2022).

5. Processor NTCz "Modul" [Processor from NTC “Modul”] URL: cnews.ru/news/top/20190306_svet_uvidel_moshchnejshij_rossijskij_nejroprotsessor (date accessed 26.03.2022).

6. Medakin P.O., Nikulin R.N., Avdeyuk O.A., Koroleva I.Yu., Pavlova E.S., Lemeshkina I.G. Inzhenernyj vestnik Dona, 2021, №1. URL: ivdon.ru/magazine/archive/n1y2021/6800/.

7. Vinokurskij D.L., Gan`shin K.Yu., Mezenceva O.S., Samojlov F.V. Inzhenernyj vestnik Dona, 2021, №9 URL: ivdon.ru/magazine/archive/n9y2021/7192/.

8. DVM-sistema razrabotki parallel`ny`x program [DVM-system for parallel program development] URL: dvm-system.org/ru/about/ (date accessed 26.03.2022).

9. Shteinberg B.Ya. Optimizaciya razmeshheniya danny`x v parallel`noj pamyati [Optimization of data placement in parallel memory]. Rostov-na-Donu: izd-vo Yuzhnogo federal`nogo universiteta, 2010, 255 p.

10. Prangishvili I.V., Vilenkin S.Ya., Medvedev I.L. Parallel`ny`e vy`chislitel`ny`e sistemy` s obshhim upravleniem [Parallel computation systems with common control]. M.: E`nergoatomizdat, 1983, 312 p.

11. Krivosheev N.M., Steinberg B.Ya. Algorithm for searching minimum inter-node data transfers. «Procedia Computer Science». 10th International Young Scientist Conference on Computational Science, YSC 2021, 1-3 July 2021, pp. 306-313.

12. Padua D., Vol`f M., Xammer K. Vektorizaciya programm: teoriya, metody`, realizaciya [Program vectorization: theory, methods and practice]. Sbornik perevodov statej M.: Mir, 1991. pp. 246 - 267.

13. Shteinberg B.Ya. Matematicheskie metody` rasparallelivaniya rekurrentny`x programmny`x ciklov dlya superkomp`yuterov s parallel`noj pamyat`yu [Mathematical methods for parallelization of recurrent program loops for supercomputers with parallel memory]. Rostov-na-Donu: izd-vo Rostovskogo universiteta, 2004, 192 p.

14. Lamport L. The parallel execution of DO loops. Commun. ACM, 1974, vol. 17, №2, pp. 83-93.

15. Steinberg O.B. Programming & Computer Software (Bulletin SUSU MMCS), Ser. Mathematical Modelling, 2020, T. 13, V. 3, pp. 59-67. DOI: doi.org/10.14529/mmp200305