УДК 004.4’422

# АВТОМАТИЗАЦИЯ РАСПАРАЛЛЕЛИВАНИЯ ПРОГРАММ ДЛЯ МНОГОЯДЕРНЫХ ПРОЦЕССОРОВ С РАСПРЕДЕЛЕННОЙ ЛОКАЛЬНОЙ ПАМЯТЬЮ

**А. П. Баглий[1] [0000-0001-9089-4164], Н. М. Кривошеев[2] [0000-0002-1366-7037], Б. Я. Штейнберг[3] [0000-0001-8146-0479]**

[1]*Южный федеральный университет, Институт математики, механики и компьютерных наук;*

[1]abagly@sfedu.ru, [2]krivosheev@sfedu.ru, [3]byshtyaynberg@sfedu.ru

***Аннотация***

В статье идет речь о создании распараллеливающих компиляторов для вычислительных систем с распределенной памятью. Промышленные распараллеливающие компиляторы распараллеливают программы для вычислительных систем с общей памятью. Преобразование последовательных программ для вычислительных систем с распределенной памятью требует разработки дополнительных функций. Это становится актуальным для перспективных систем на кристалле с сотнями и более ядер. В терминах графа информационных связей сформулировано условие распараллеливания программного цикла на вычислительную систему с распределенной памятью.

***Ключевые слова**: автоматизация распараллеливания, распределенная память, преобразования программ, размещение данных, пересылки данных*

## ВВЕДЕНИЕ

Во многих публикациях рассматривается задача автоматического создания параллельного кода для систем с распределенной память, но предлагаются полуавтоматические инструменты, в которых пользователь должен вызывать специальные функции или писать директивы компилятору. В работе [4] отмечено, что автоматическая компиляция последовательной программы для параллельной архитектуры с распределенной памятью является очень сложной задачей, не имеющей в настоящее время эффективного решения. В этой же работе описана генерация параллельного кода с генерацией коммуникаций, основанных на MPI (для

кластера с мультипроцессорами). Об оптимизации размещения данных не говорится ничего – значит, пользователь компилятора должен это делать сам. Использована полиэдральная распараллеливающая система Pluto, которая позволяет в пространстве итераций находить подмножества точек, допускающих параллельное выполнение. Ничего не говорится о локализации данных, хотя на современных вычислительных архитектурах распараллеливание эффективно при параллельном выполнении относительно больших фрагментов кода [16], особенно при распределенной памяти.

Для вычислительной системы (ВС) с распределенной памятью самой длительной операцией является межпроцессорная пересылка данных. Как показано в [8], накладные расходы, связанные с пересылками данных, приходится учитывать при отображении программ на распределенную память. Но в последнее время появляются многоядерные процессоры, иногда называемые «суперкомпьютер на кристалле», с десятками, сотнями и тысячами ядер [9, 10, 13]. Пересылка данных между процессорными ядрами на одной микросхеме требует значительно меньше времени, чем на коммуникационной сети (Ethernet, Infiniband, PCI-express, …). Это означает расширение множества эффективно распараллеливаемых программ и делает целесообразным разработку распараллеливающих компиляторов. К этому списку можно добавить ориентированные на быструю работу с нейронными сетями процессоры, которые появляются в последние несколько лет [18–24].

В [14, 15] описаны блочно-аффинные размещения данных в распределенной памяти. Следует отметить работы группы DVM-System [5, 11] по генерации параллельного кода на ВС с распределенной памятью, где спецификации параллелизма (DVMH-указания) оформляются в виде специальных комментариев. Генерирующие автоматически параллельный код на ВС с распределенной памятью компилирующие системы DVM, Parawise и др. предполагают дописывание текста последовательной программы без предварительного преобразования. В работе [8] рассмотрена задача трансляции высокоуровневых описаний параллельной обработки данных в программе на уровень конструкций стандарта MPI для выполнения на системе с распределенной памятью. В [7] рассмотрена задача распарал-

леливания программного цикла на ВС с распределенной памятью с минимизацией межпроцессорных пересылок. Время, необходимое на пересылки данных, нелинейно зависит от объема пересылаемых данных. Значительное время отнимает инициализация пересылки. Метод размещения данных с перекрытиями [1, 3] существенно ускоряет параллельные итерационные алгоритмы за счет уменьшения количества пересылок при укрупнении множеств пересылаемых данных.

Для многих алгоритмов удобными являются циклические пересылки данных – это такие пересылки, которые для некоторой целой константы $C$ из каждого процессорного элемента с номером $k$ пересылают данные в процессорный элемент с номером $(p + C)\ mod\ p$, где $p$ – количество процессорных элементов. Многие коммуникационные сети (например, кольцевая шина, mesh или тор) позволяют выполнять циклические пересылки для всех $k$ за один такт. В [12] приведено много задач линейной алгебры и математической физики, для параллельного решения которых на ВС с распределенной памятью используются циклические пересылки.

В [2] приведены результаты экспериментов, показывающие, что современные оптимизирующие компиляторы плохо оптимизируют код и имеют большой неиспользованный потенциал оптимизирующих преобразований. Можно полагать, что для микросхем с сотнями вычислительных ядер этот потенциал оптимизаций больше, чем для процессоров, на которых проводились эксперименты.

Данная работа направлена на создание распараллеливающего компилятора, который автоматически анализирует высокоуровневый текст программы, находит размещение данных в распределенной памяти с минимизацией межпроцессорных пересылок, выполняемых коммуникационной сетью, и в итоге преобразует программу к виду, допускающему распараллеливание на ВС с распределенной памятью. Ниже приведен пример с генерацией параллельного MPI-кода, хотя можно использовать SHMEM или другие инструменты.

Настоящая работа отличается от других попыток автоматизировать отображение последовательных программ на вычислительные архитектуры с распределенной памятью тем, что в ней автоматически выполняется размещение массивов в распределенной памяти по тексту входной программы, написанному на высокоуровневом языке, а не директивам, дописанным к программе вручную. Это

удается сделать благодаря тому, что оптимальные размещения массивов ищутся среди разработанных ранее блочно-аффинных размещений массивов [14, 15], которые, с одной стороны, могут быть описаны малым количеством параметров (пропорционально размерности массива), а, с другой стороны, покрывают размещения, широко используемые на практике. Расширение множества распараллеливаемых программ может быть достигнуто с помощью оптимизирующих преобразований программ, которые имеются в используемой нами оптимизирующей распараллеливающей системе (ОРС) и известных оптимизирующих компиляторах LLVM, GCC, ICC, MS-compiler.

## 1. РАСПАРАЛЛЕЛИВАЕМЫЕ ПРОГРАММНЫЕ ЦИКЛЫ

Будем рассматривать задачу параллельного выполнения цикла. Как обычно, под распараллеливанием цикла понимаем одновременное выполнение его итераций – но это определение предполагает уточнения.

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

Листинг 1: Простой цикл

Будем полагать, что цикл удовлетворяет следующим условиям.

1. В теле цикла счетчик цикла j не изменяет значения; имеется только один выход из цикла 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);
2. В цикле есть только вхождения одномерных массивов, индексное выражение которых имеет вид `(j + k)`, где `j` – счетчик цикла, k – некоторая константа или переменная, не изменяющая своего значения (в теле цикла);
3. В цикле есть только операторы присваивания.

*Замечание.* В последующих работах представленные ограничения будут ослаблены.

## 2. БЛОЧНО-АФФИНЫЕ РАЗМЕЩЕНИЯ МАССИВОВ

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, получение данных из которого происходит быстрее, чем из модулей памяти других ПЭ. Все ПЭ занумерованы, начиная с нуля. Размещение массива в памяти – это функция, которая для каждого элемента массива возвращает 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 (включая нуль) числа $p, d_1, d_2, \dots, d_m$ и целые константы $s_0, s_1, s_2, \dots, s_m$ зависят только от $m$-мерного массива $X$. Блочно-аффинное по модулю $p$ размещение $m$-мерного массива $X$ – это такое размещение, при котором элемент $X[i_1, i_2, \dots, i_m]$ находится в модуле памяти ПЭ с номером $u = \left(\left\lfloor\frac{i_1}{d_1}\right\rfloor * s_1 + \left\lfloor\frac{i_2}{d_2}\right\rfloor * s_2 + \dots + \left\lfloor\frac{i_m}{d_m}\right\rfloor * s_m + s_0\right) \ mod\ p.$

Число $s_0$ показывает номер модуля памяти, в котором размещается нулевой элемент $X[i_1, i_2, \dots, i_m]$. При описанном блочно-аффинном способе размещения $m$-мерный массив представляется как массив блоков размерности $d_1 * d_2 * \dots * d_m$, который размещается так, что у каждого блока все элементы оказываются в модуле памяти одного ПЭ. Числа $p, d_1, d_2, \dots, d_m, s_0, s_1, s_2, \dots, s_m$ будем называть параметрами размещения.

## 3. МЕЖПРОЦЕССОРНЫЕ ПЕРЕСЫЛКИ ДАННЫХ

Пересылка – это команда коммуникационной системы. Шина и кольцо позволяют выполнять межпроцессорные циклические пересылки. В данной статье будем рассматривать только циклические пересылки данных. К ним сводятся параллельные алгоритмы для многих задач линейной алгебры и численных мето-

дов решения задач математической физики [12]. Актуальность именно таких пересылок для ВС близкого будущего подчеркнута в [6].

Опишем условия, при которых можно в текст программного цикла вставить пересылку.

Рассмотрим последовательно выполняемый оператор цикла, содержащий два блока, между которыми планируется вставка пересылки

```
for (int j = 0; j < N; ++j) {
  B1(j);
  B2(j);
}
```

После вставки пересылки этот оператор цикла будет иметь вид

```
for (int j = 0; j < N; ++j) {
  B1(j);
  XX[j] ← X[j + k];
  B2(j);
}
```

Здесь пересылка в последовательной программе означает присваивание элементам нового массива элементов старого массива. Поскольку при переходе к параллельному выполнению цикла пересылка должна выполняться одновременно для всех значений счетчика цикла, этот код должен быть эквивалентен следующему

```
for (int j = 0; j < N; ++j) {
  B1(j);
}
for (int j = 0; j < N; ++j) {
  XX[j] ← X[j + k];
}
for (int j = 0; j < N; ++j) {
  B2(j);
}
```

В частности, отсюда вытекает, что к последовательному циклу можно применить преобразование «разрезание цикла». Это означает, что в исходном последовательном цикле не должно быть дуг графа информационных связей «снизу-вверх», точнее, из блока B1 в блок B2 [17]. Это требование можно ослабить, если вставку пересылки сопровождать переименованием некоторых вхождений пере-

менной X, заменив эту переменную новой переменной XX. Такое переименование позволяет иметь в исходном цикле дуги «снизу-вверх», которые ведут из вхождений переменной X нижнего блока B2 в верхний блок B1 и которые предполагается заменить новой переменной XX.

Заметим, в частности, что после вставки пересылки переименования вхождений переменной X возможны только в блоке B2, но не в B1.

**Пример 1.** В рассматриваемом цикле есть только одна дуга графа информационных связей, ведущая «снизу-вверх» от вхождения `X[j + 2]` к вхождению-генератору `X[j]`.

```
for  (int j = 0; j < N; ++j) {
  X[j] = A[j] + B[j] * X[j];
  Y[j] = X[j] + X[j + 2] * C[j + 1];
}
```

После вставки пересылки дуга графа информационных связей, ведущая «снизу-вверх», исчезает из-за переименования.

```
for (int j = 0; j < N; ++j) {
  X[j] = A[j + 3] + B[j] * X[j];
  XX[j] ← X[j + 2];
  Y[j] = X[j] + XX[j] * A[j + 1];
}
```

Условие распараллеливания программного цикла на вычислительную систему с распределенной памятью со вставкой межпроцессорных пересылок состоит в отсутствии дуг графа информационных связей «снизу-вверх», кроме дуг, ведущих из вхождений переменных, к которым применяется переименование (такие дуги исчезают после переименования). Такое переименование невозможно для дуг истинной или выходной зависимостей, а возможно только для антизависимостей.

Алгоритм распараллеливания программного цикла на вычислительную систему с распределенной памятью со вставкой межпроцессорных пересылок должен начинаться с приведения операторов тела цикла к виду, в котором все дуги графа информационных связей после соответствующих переименований будут направлены «сверху-вниз». Это возможно достичь в несколько этапов.

1) Избавляемся от выходных информационных зависимостей.

2) Располагаем операторы тела цикла так, чтобы все дуги истинной зависимости были направлены сверху вниз. Если это невозможно, то программный цикл имеет рекуррентно вычисляемые переменные и не может быть распараллелен.

3) Ищем наименьшее множество пересылок, которые необходимо вставить при распараллеливании.

4) Вставляем найденные пересылки в текст программного цикла.

5) Выполняем необходимые переименования вхождений переменных, которые копированы при пересылках.

**Замечание.** Переименования могут быть не связаны с удалением дуг графа информационных связей «снизу-вверх», например, переименования могут быть связаны с разрывом циклов ГОП (графа операторы-переменные).

**Пример 2.** В рассматриваемом цикле есть только одна дуга истинной зависимости графа информационных связей, ведущая «снизу-вверх» от вхождения `Y[j]` к вхождению-генератору `Y[j - 1]`.

```
for (int j = 1; j < N; ++j) {
  X[j] = A[j] * Y[j - 1] + Y[j + 1] * X[j];
  Y[j] = X[j] + X[j + 2] * C[j + 1];
  A[j] = Y[j + 1] * X[j];
}
```

В первую очередь переставим второй оператор присваивания на первое место, чтобы истинную зависимость по переменной Y, которая вела «снизу-вверх», направить в обратную сторону.

```
for (int j = 1; j < N; ++j) {
  Y[j] = X[j] + X[j + 2] * C[j + 1];
  X[j] = A[j] * Y[j - 1] + Y[j + 1] * X[j];
  A[j] = Y[j + 1] * X[j];
}
```

Теперь можно вставлять пересылки данных. Приведем один из вариантов таких вставок.

```
for (int j = 1; j < N; ++j) {
  XX[j] ← X[j + 2];
  Y[j] = X[j + 1] + XX[j] * C[j + 1];
  YY[j] ← Y[j + 1];
  AA[j] ← A[j + 3];
  X[j] = A[j] * Y[j - 1] + YY[j] * AA[j];
  A[j] = YY[j] * X[j];}
```

**Определение 2.** Размещение переменных будем называть согласованным для данного оператора, если для каждого значения счетчика цикла все вхождения переменных, входящих в данный оператор, расположены в одном и том же ПЭ.

Ясно, что для параллельного выполнения цикла на ВС с распределенной памятью оператор цикла и, в частности, все операторы цикла должны быть согласованы. Согласованность операторов может быть достигнута с помощью межпроцессорных пересылок и блочно-аффинных размещений данных.

## 4. ВЫБОР ОПТИМАЛЬНОГО РАЗМЕРА БЛОКА В БЛОЧНО-АФФИННОМ РАЗМЕЩЕНИИ ДАННЫХ

В данной работе рассматривается такой программный цикл, в котором шаг равен 1, тело содержит только операторы присваивания, правая часть каждого оператора присваивания является выражением, содержащим только вхождения массивов, левая часть каждого оператора присваивания является вхождением массива, а все вхождения массивов имеют вид a[i + c], где a – имя некоторого массива, i – счетчик цикла, c – некоторая целочисленная константа, известная на этапе компиляции.

```
{
  int i;
  for (…; …; ++i) {
    a_{j_1}[i + c_1] = f_1(a_{j_{1,1}}[i + c_{1,1}], …, a_{j_{1,n_1}}[i + c_{1,n_1}]);
    …
    a_{j_k}[i + c_k] = f_k(a_{j_{k,1}}[i + c_{k,1}], …, a_{j_{k,n_k}}[i + c_{k,n_k}]);
  }
}
```

Листинг 2: Вид программного цикла

Будем считать, что цикл выполняет N итераций, тогда можно выполнить гнездование цикла с размером блока d, то есть преобразовать его к виду

```
{
  int i, j;
  for (j = 0; j  < N; j += d) {
    for (i = j; i < min(N, j + d); ++i) {
      a_{j_1}[i + c_1] = f_1(a_{j_{1,1}}[i + c_{1,1}], …, a_{j_{1,n_1}}[i + c_{1,n_1}]);
```

…

$a_{j_k}[i + c_k] = f_k\left(a_{j_{k,1}}[i + c_{k,1}], \dots, a_{j_{k,n_k}}[i + c_{k,n_k}]\right);$

}

}

}

Листинг 3: Вид программного цикла после гнездования

Пусть каждый процессорный элемент целиком выполняет итерации внешнего цикла, приведенного на листинге 3, причем $i$-ую итерацию выполняет процессорный элемент с номером $\left\lfloor \frac{i}{d} \right\rfloor \ mod\ p$, где $d$ – размер блока, $p$ – число процессорных элементов. Пусть массивы размещены без дублирования, то есть каждый элемент массива размещен только в ПЭ.

**Определение 3.** Одинарная пересылка — это пересылка одного данного из одного ПЭ в другой ПЭ.

Следует отметить, что циклическая пересылка представляет собой множество одинарных пересылок, выполняемых параллельно.

В данной работе мы рассматриваем задачу поиска минимума одинарных пересылок. Минимум пересылок с учетом свойств коммуникационной сети не превосходит минимума одинарных пересылок.

**Пример 3.** Рассмотрим программный цикл

```
for (int i = 0; i < 10; ++i) {
  a[i] = b[i + 2];
  c[i] = c[i - 1];
}
```

Пусть параметр гнездования цикла равен 2, тогда после гнездования цикл примет вид

```
for (int j = 0; j < 10; j += 2) {
  for (int i = j; i < j + 2; ++i) {
    a[i] = b[i + 2];
    c[i] = c[i - 1];
  }
}
```

Пусть процессорных элементов $p = 3$, размер блока $d = 2$, элементы массива a размещены $P_a(i) = 2$ (все в одном ПЭ с номером 2), элементы массива b размещены $P_b(i) = (i\ mod\ p)$, элементы массива c размещены $P_c(i) =$

$\left(\left\lfloor\frac{i+1}{d}\right\rfloor\ mod\ p\right)$. Тогда вхождение a[i] требует одинарную пересылку на итерации I = 2 исходного цикла, но не требует одинарную пересылку на итерации 4, вхождение b[i + 2] требует 7 одинарных пересылок, а вхождение c[i] требует одинарную пересылку на каждой итерации вида $1 + 2\ *\ k$, где $k \in \mathbb{Z}$.

В примере 3 массив a размещен блочно-аффинно $P_a(i) = 2 = \left(\left\lfloor\frac{1}{d_1}\right\rfloor * i + 2\right)\ mod\ p$, где $d_1$ – большое число (не меньшее количества итераций цикла), $s_0 = 2, s_1 = 1;\ P_b(i) = (i\ mod\ p)$ определяет блочно-аффинное размещение с $s_b = 0, d = 1, P_c(i) = \left(\left\lfloor\frac{i+1}{d}\right\rfloor\ mod\ p\right)$ определяет блочно-аффинное размещение с $s_c = -1$.

Пусть массив a размещен блочно-аффинно. Пусть вхождение a[i + c] требует (не требует) пересылки на итерации $i_0$, тогда на любой итерации вида $i_0 + d * k$, где $k \in \mathbb{Z}$, это вхождение также требует (не требует) пересылки (здесь $d$ – размер блока).

**Теорема.** *Пусть количество процессорных элементов* $p > 1 + \max\{|c_{i,j}|\}$, *где* $c_{i,j}$ *– константы в индексных выражениях цикла, приведенного в листинге 2, тогда минимальное количество одинарных пересылок достигается при размере блока* $d = \left\lfloor\frac{N}{p}\right\rfloor$.

Минимизация циклических пересылок для размещений массивов с блоками размера 1 рассмотрена в [7].

## 5. СОЗДАНИЕ ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ С ПОМОЩЬЮ РАЗМЕЩЕНИЙ И ПЕРЕСЫЛОК

При заданном размещении массивов для гнезда циклов с пересылками можно построить эквивалентную параллельную программу. Для параллельного выполнения можно использовать средства MPI. Размещения массивов по ПЭ, пересылки блоков элементов массивов и распределение итераций циклов между ПЭ отображаются в вызовы функций MPI, объявление вспомогательных массивов и эквивалентные преобразования циклов. Рассмотрим, с помощью каких конструкций и преобразований реализуется параллельная программа. Для описания

блочно-аффинного размещения массива A с заданными параметрами потребуются заведение дополнительного массива, описание пользовательского типа и вызов коллективной операции распределения элементов массива по всем ПЭ. Пользовательский тип служит для задания подмножества элементов массива, которые отправляются в один ПЭ. Тип создается с помощью функций MPI_Type_vector и MPI_Type_contiguous. Например, рассылка элементов в p узлов из одномерного массива X в размещенный на ПЭ массив XX проводится инструкцией

```
MPI_Scatter(X - 2, 1, T, XX + rank - 2, 1, T, 0, MPI_COMM_WORLD);
```

где T – пользовательский тип. Для удобства работы со сдвигами внутри массива здесь предполагается, что массив X – это указатель (на языке C), который указывает внутрь массива большего размера, в котором гарантированно помещаются все нужные элементы. После завершения цикла потребуется собрать элементы обратно в массив, размещенный в главном узле, с помощью операции MPI_Gather

```
MPI_Gather(YY + rank, 1, T, Y, 1, T, 0, MPI_COMM_WORLD);
```

Пересылки элементов массивов внутри цикла возможно реализовать с помощью вызова операции MPI_Sendrecv, в которой используются аналогичные пользовательские типы, построенные с учетом размещений двух массивов, участвующих в пересылке. Для организации пересылки по кольцу ПЭ ранги отправителя и получателя вычисляются с учетом расстояния пересылки.

**Пример 4.** Распределение итераций цикла между ПЭ можно описать как два последовательных преобразования:

- Гнездование цикла;
- Удаление заголовка внешнего цикла с установкой значения счетчика, равного рангу ПЭ.

Рассмотрим элементарный цикл, к которому можно применить этот подход.

```
for (int j = 0; j < N; ++j) {
  Y[j] = (XX[j - 1] + X1[j] + X2[j + 1]) / 3.0;
}
```

Для параллельного выполнения этот цикл преобразуется к виду

```
for (int j2 = 0; j2 < 11 / p; ++j2) {
  int j = j2 * p + j1 + padding;
  MPI_Sendrecv(XX + rank, 1, T, rank - 1, 0, X1 + rank - 1,
  1, T, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
  MPI_Sendrecv(XX + rank, 1, T, rank - 2, 0, X2 + rank, 1,
  T, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
  Y[j] = (XX[j - 1] + X1[j] + X2[j + 1]) / 3.0;
}
```

Листинг 4. Цикл после применения всех преобразований

В этом примере дополнительные массивы X1, X2 размещены аналогично с параметром $s_0 = 1$ и $0$ соответственно, массивы Y и XX размещены с параметром $s_0 = 1$ и $2$. Такое размещение согласовано. Пример служит для демонстрации генерации кода с использованием MPI.

## 6. ДАЛЬНЕЙШИЕ ИССЛЕДОВАНИЯ

В дальнейших работах предполагается рассмотреть многомерные циклы с многомерными массивами и с аппаратной поддержкой не только циклических сдвигов, но и операции широковещательной рассылки данных.

В данной статье для распараллеливания программного цикла использовалось преобразование «гнездование цикла». В последующих работах предполагается использовать и некоторые другие преобразования для расширения множества эффективно распараллеливаемых программ.

## 7. ЗАКЛЮЧЕНИЕ

Разработано преобразование программного цикла с автоматическим размещением данных в распределенной памяти и минимизацией межпроцессорных пересылок. Статья представляет собой шаг на пути к созданию оптимизирующих распараллеливающих компиляторов на высокопроизводительные системы на кристалле нового поколения типа «суперкомпьютер на кристалле».

**Благодарности**

Исследование выполнено при финансовой поддержке гранта Российского научного фонда № 22-21-00671, https://rscf.ru/project/22-21-00671/

**СПИСОК ЛИТЕРАТУРЫ**

1. *Ammaev S.G., Gervich L.R., Steinberg B.Y.* Combining parallelization with overlaps and optimization of cache memory usage // PaCT 2017: Parallel Computing Technologies, Lecture Notes in Computer Science. 2017. Vol. 10421. P. 257–264.

2. *Gong Z., Chen Z., Szaday Z., Wong D., Sura Z., Watkinson N., Maleki S., Padua D., Veidenbaum A., Nicolau A.* An empirical study of the effect of source-level loop transformations on compiler stability // Proceedings of the ACM on Programming Languages. 2018. P. 1–29.

3. *Гервич Л.Р., Кравченко Е.Н., Штейнберг Б.Я., Юрушкин М.В.* Автоматизация распараллеливания программ с блочным размещением данных // Сибирский журнал вычислительной математики. 2015. Т. 18. №1. С. 41–53.

4. *Bondhugula U.* Automatic distributed-memory parallelization and codegeneration using the polyhedral framework // Technical report ISc-CSA-TR-2011-3. 2011. 10 p.

5. DVM-система разработки параллельных программ. URL: http://dvm-system.org/ru/about/, дата обращения 26.03.2022.

6. *Корнеев В.В.* Параллельное программирование // Программная инженерия. 2022. Т. 13. № 1. С. 3–16.

7. *Krivosheev N.M., Steinberg B.Ya.* Algorithm for searching minimum internode data transfers. // «Procedia Computer Science», 10th International Young Scientist Conference on Computational Science. YSC 2021. 1–3 July 2021. P. 306–313.

8. *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. P. 152–157.
https://doi.org/10.1109/ISPAN.1999. 778932.

9. Epiphany-V: A 1024-core 64-bit RISC processor.
URL: https://parallella.org/2016/10/05/epiphany-v-a-1024-core-64-bit-risc-processor, дата обращения 26.03.2022.

10. SoC Esperanto. URL: https://www.esperanto.ai/technology, дата обращения 26.03.2022.

11. *Бахтин В.А., Захаров Д.А., Колганов А.С., Крюков В.А., Поддерюгина Н.В., Притула М.Н.* Решение прикладных задач с использованием DVM-системы //

Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8. № 1. С. 89–106.

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

13. Процессор НТЦ «Модуль».
URL: https://www.cnews.ru/news/top/2019-03-06_svet_uvidel_moshchnejshij_rossijskij_nejroprotsessor, дата обращения 26.03.2022.

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

15. *Штейнберг Б.Я.* Оптимизация размещения данных в параллельной памяти. Ростов-на-Дону: Изд-во Южного федерального университета, 2010. 255 с.

16. *Vasilenko A., Veselovskiy V., Metelitsa E., Zhivykh N., Steinberg B., Steinberg O.* Precompiler for the ACELAN-COMPOS Package Solvers // In: Malyshkin V. (eds). Parallel Computing Technologies. PaCT 2021. Lecture Notes in Computer Science. Vol. 12942. P. 103–116. Springer, Cham.
https://doi.org/10.1007/978-3-030-86359-3_8

17. *Штейнберг О.Б.* Минимизация количества временных массивов в задаче разбиения циклов // Известия ВУЗов. Северо-Кавказский регион. Естественные науки. 2011. №5. С. 31–35.

18. SambaNova Launches Second-Gen DataScale System.
URL: https://www.hpcwire.com/2022/09/14/sambanova-launches-second-gen-datascalesystem, дата обращения 20.01.2023.

19. *Елизаров Г.С., Конотопцев В.Н., Корнеев В.В.* Специализированные большие интегральные схемы для реализации нейросетевых выводов // XXII международная конференция «Харитоновские тематические научные чтения «Суперкомпьютерное моделирование и искусственный интеллект»: сб. трудов / Под ред. Р.М. Шагалиева. Саров: ФГУП «РФЯЦ-ВНИИЭФ», 2022. С. 181–184.

20. *Корнеев В.В.* Направления повышения производительности нейросетевых вычислений // Программная инженерия. 2020. Т. 11, № 1. С. 21–25.
https://doi.org/10.17587/prin.11.21–25

21. *Yen I.E., Xiao Zh., Xu D.* S4: a High-sparsity, High-performance AI Accelerator // arXiv:2207.08006v1 [cs.AR] 16 Jul 2022

22. *Gale T., Elsen E., Hooker S.* The state of sparsity in deep neural networks // arXiv preprint arXiv:1902.09574, 2019

23. Intelligence Processing Unit. URL: https://www.graphcore.ai/products/ipu. (accessed: 20.01.2023)

24. *Jia Zh., Tillman B., Maggioni M., Scarpazza D.P.* Dissecting the Graphcore IPU Architecture via Microbenchmarking // Technical Report. December 7, 2019. arXiv:1912.03413v1 [cs.DC] 7 Dec. 2019. 91 p.

## AUTOMATION OF PROGRAM PARALLELIZATION FOR MULTICORE PROCESSORS WITH DISTRIBUTED LOCAL MEMORY

**A. P. Bagliy[1] [0000-0001-9089-4164], N. M. Krivosheev[2] [0000-0002-1366-7037],**
**B. Ya. Steinberg[3] [0000-0001-8146-0479]**

[1]*Southern federal university, Faculty of mathematics, mechanics and computer science*

[1]abagly@sfedu.ru, [2]krivosheev@sfedu.ru, [3]byshtyaynberg@sfedu.ru

**Abstract**

This paper is concerned with development of parallelizing compiler onto computer system with distributed memory. Industrial parallelizing compilers create programs for shared memory systems. Transformation of sequential programs onto systems with distributed memory requires development of new functions. This is becoming topical for future computer systems with hundreds and more cores. Conditions for program loop parallelization onto computer system with distributed memory is formulated in terms of information dependence graph.

***Keywords****: automatic parallelization, distributed memory, program transformation, data distribution, data interchange*

**REFERENCES**

1. *Ammaev S.G., Gervich L R., Steinberg B.Y.* Combining parallelization with overlaps and optimization of cache memory usage // PaCT 2017: Parallel Computing Technologies, Lecture Notes in Computer Science. 2017. Vol. 10421. P. 257–264.

2. *Gong Z., Chen Z., Szaday Z., Wong D., Sura Z., Watkinson N., Maleki S.,*

*Padua D., Veidenbaum A., Nicolau A.* An empirical study of the effect of source-level loop transformations on compiler stability // Proceedings of the ACM on Programming Languages. 2018. P. 1–29.

3. *Gervich L.R., Kravchenko E.N., Steinberg B.Y., Yurushkin M.V.* Automatic program parallelization with block data distribution // Sibirskiy zhurnal vychislitelnoi matematiki. 2015. Vol. 18. No. 1. P. 41–53.

4. *Bondhugula U.* Automatic distributed-memory parallelization and codegeneration using the polyhedral framework // Technical report ISc-CSA-TR-2011-3. 2011. 10 p.

5. DVM-sistema razrabotki parallel'nyh program.
URL: http://dvm-system.org/ru/about, last accessed 26.03.2022.

6. *Korneev V.V.* Parallel'noe programmirovanie // Programmnaya Ingeneria. 2022. Vol. 13. No. 1. P. 3–16.

7. *Krivosheev N.M., Steinberg B.Ya.* Algorithm for searching minimum internode data transfers // «Procedia Computer Science», 10th International Young Scientist Conference on Computational Science. YSC 2021. 1–3 July 2021. P. 306–313.

8. *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. P. 152–157.
https://doi.org/10.1109/ISPAN.1999. 778932.

9. Epiphany-V: A 1024-core 64-bit RISC processor.
URL: https://parallella.org/2016/10/05/epiphany-v-a-1024-core-64-bit-risc-processor, last accessed 26.03.2022.

10. SoC Esperanto. URL: https://www.esperanto.ai/technology, last accessed 26.03.2022.

11. *Bahtin V.A., Zaharov D.A., Kolganov A.S., Kryukov V.A., Podderyugina N.V., Pritula M.N.* Reshenie prikladnyh zadach s ispol'zovaniem DVM-sistemy // Vestnik YUUrGU. Seriya: Vychislitel'naya matematika i informatika. 2019. T.. 8. № 1. S. 89–106.

12. *Prangishvili I.V., Vilenkin S.YA., Medvedev I.L.* Parallel'nye vychislitel'nye sistemy s obshchim upravleniem. M.: Energoatomizdat, 1983. 312 p.

13. Processor NTC "Modul". URL: https://www.cnews.ru/news/top/2019-03-06_svet_uvidel_moshchnejshij_rossijskij_nejroprotsessor, last accessed 26.03.2022.

14. *Shtejnberg B.Ya.* Blochno-affinnye razmeshcheniya dannyh v parallel'noj pamyati // Informacionnye tekhnologii. 2010. №6. S. 36–41.

15. *Shtejnberg B.Ya.* Optimizaciya razmeshcheniya dannyh v parallel'noj pamyati. Rostov-na-Donu: Izd-vo Yuzhnogo federal'nogo universiteta, 2010. 255 s.

16. *Vasilenko A., Veselovskiy V., Metelitsa E., Zhivykh N., Steinberg B., Steinberg O.* Precompiler for the ACELAN-COMPOS Package Solvers // In: Malyshkin V. (Ed.) Parallel Computing Technologies. PaCT 2021. Lecture Notes in Computer Science. Vol. 12942. P. 103-116. Springer, Cham. https://doi.org/10.1007/978-3-030-86359-3_8

17. *Shtejnberg O.B.* Minimizaciya kolichestva vrevtnnyh massivov v zadache raszbieniya ziklov // Izvestia VUZov. Severo-Kavkazsky region. Estestvennye nauki. 2011. №5. S. 31–35

18. SambaNova Launches Second-Gen DataScale System.
URL: https://www.hpcwire.com/2022/09/14/sambanova-launches-second-gen-datascalesystem, last accessed 20.01.2023.

19. *Elizarov G.S., Konotoptsev V.N., Korneev V.V.* Specialized large integrated circuits for the implementation of neural network inference // XXII International conference "Kharitonov thematic scientific readings "Supercomputer modeling and Artificial Intelligence": proceedings / Edited by R.M. Shagaliev. Sarov: FSUE "RFSC-VNIIEF", 2022. P. 181–184 (in Russian).

20. *Korneev V.V.* Approaches to improving the performance of neural network computing // Programmnaya ingeneria. 2020. Vol. 11. No. 1. P. 21–25.
https://doi.org/10.17587/prin.11.21-25 (in Russian).

21. *Yen I.E., Xiao Zh., Xu D.* S4: a High-sparsity, High-performance AI Accelerator // arXiv:2207.08006v1 [cs.AR] 16 Jul 2022

22. *Gale T., Elsen E., Hooker S.* The state of sparsity in deep neural networks // arXiv preprint arXiv:1902.09574, 2019

23. Intelligence Processing Unit. URL: https://www.graphcore.ai/products/ipu, last accessed: 20.01.2023.

24. *Jia Zh., Tillman B., Maggioni M., Scarpazza D.P.* Dissecting the Graphcore IPU Architecture via Microbenchmarking // Technical Report. December 7, 2019. arXiv:1912.03413v1 [cs.DC] 7 Dec 2019. 91 p.

## СВЕДЕНИЯ ОБ АВТОРАХ

***БАГЛИЙ Антон Павлович*** – ст. преподаватель Института математики, механики и компьютерных наук Южного федерального университета

***Anton Pavlovich BAGLIY*** – senior teacher in department of Mathematics, mechanics and computer science of Southern federal university.

email: abagly@sfedu.ru

ORCID 0000-0001-9089-4164

***КРИВОШЕЕВ Никита Максимович*** – студент 1 курса магистратуры Института математики, механики и компьютерных наук Южного федерального университета.

***Nikita Maksimovich KRIVOSHEYEV*** – first-year master's student in department of Mathematics, mechanics and computer science of Southern federal university.

email: krivosheev@sfedu.ru

ORCID 0000-0002-1366-7037

***ШТЕЙНБЕРГ Борис Яковлевич*** – д. т. н, зав. каф., с. н. с. Института математики, механики и компьютерных наук Южного федерального университета.

***Boris Yakovlevich STEINBERG*** – doctor of computer science, head of chair in department of Mathematics, mechanics and computer science of Southern federal university.

email: borsteinb@mail.ru

ORCID: 0000-0001-8146-0479

*Материал поступил в редакцию 20 января 2023 года*