ИПМ им.М.В.Келдыша РАН

Абрау-2022 • Труды конференции

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

Автоматизация распараллеливания программ с оптимизацией пересылок данных

Рекомендуемая форма библиографической ссылки

Баглий А.П., Кривошеев Н.М., Штейнберг Б.Я. Автоматизация распараллеливания программ с оптимизацией пересылок данных // Научный сервис в сети Интернет: труды XXIV Всероссийской научной конференции (19-22 сентября 2022 г., онлайн). — М.: ИПМ им. М.В.Келдыша, 2022. — С. 81-92.
https://doi.org/10.20948/abrau-2022-17
https://keldysh.ru/abrau/2022/theses/17.pdf

Видеозапись выступления

Автоматизация распараллеливания программ с оптимизацией пересылок данных

А. П. Баглий[1], Н. М. Кривошеев,[1], Б. Я. Штейнберг[1]

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

Аннотация. В данной работе 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 актуально для перспективных систем на кристалле с сотнями и более ядер.

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

Automation of programs parallelization with optimization of data transfers

A. P. Bagliy[1], N. M. Krivosheev[1], B. Ya. Steinberg[1]

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

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.

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

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] отмечается, 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 описывается 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. Используется полиэдральная распараллеливающая система Pluto, которая позволяет в пространстве итераций находить подмножества точек, допускающий 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 ВС (вычислительной системы) 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 [10], [13], [9]. 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.

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 прагм в текст 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 работе [7] рассмотрена задача распараллеливания программного цикла на ВС с распределенной памятью и 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 [6]. Циклические пересылки данных – это такие пересылки, которые для некоторой целой константы C из каждого процессорного элемента с номером k пересылают данное в процессорный элемент с номером (k+C) mod p, где p – количество процессорных элементов. Многие коммуникационные сети (например, кольцевая шина, mesh или top) позволяют выполнять циклические пересылки для всех k за один такт. В [12] приводится много задач линейной алгебры и задач математической физики, для параллельного решения которых на ВС с распределенной памятью используются циклические пересылки.

В [2] приводятся результаты экспериментов, показывающие, что современные оптимизирующие компиляторы плохо оптимизируют код, 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. В данной работе приводится пример с генерацией параллельного MPI-кода, хотя можно использовать SHMEM или другие инструменты.

Данная работа отличается от других попыток автоматизировать отображение последовательных программ на вычислительные архитектуры с распределенной памятью тем, что в данной работе автоматически выполняется размещение массивов в распределенной памяти по написанному на высокоуровневом языке тексту входной программы, а не по вручную дописанным к программе директивам. Это удается благодаря тому, что оптимальные размещения массивов ищутся среди разработанных ранее блочно-аффинных размещений массивов [14], [15], которые, с одной стороны, описываются малым количеством параметров (пропорционально размерности массива), а, с другой стороны, покрывают широко используемые на практике размещения. Расширение множества распараллеливаемых программ может быть достигнуто с помощью оптимизирующих преобразований программ, которые имеются в используемой авторами данной статьи ОРС (Оптимизирующая распараллеливающая система), и в 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 меняет значения; из цикла только один выход после завершения всех итераций и переход с 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 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);
3. В цикле есть только операторы присваивания.

Замечание. В последующих работах представленные ограничения будут ослаблены.

2. Блочно-аффинные 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 ноль) числа p, d1, d2, ..., dm и целые константы s0, s1, s2, ..., sm зависят только от m-мерного массива X. Блочно-аффинное по модулю p размещение m-мерного массива X – это такое размещение, при котором элемент X[i1, i2, ..., im] находится в модуле памяти ПЭ с номером

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

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 $d_1 * d_2 * ... * d_m$, 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.

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. К таким пересылкам данных сводятся параллельные алгоритмы для многих задач линейной алгебры и численных методов решения задач математической физики [12]. Актуальность именно таких пересылок для ВС близкого будущего подчеркивается в [6].

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 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 = i + 1) {
    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 = j + d) {
    for (i = j; i < min(N, j + d); i = i + 1) {
      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 3: Вид программного цикла после гнездования

Пусть каждый процессорный элемент целиком выполняет итерации внешнего цикла, приведенного на листинге 3, причем -ую итерацию выполняет процессорный элемент с номером , 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 рассматривается задача поиска минимума одинарных пересылок. Минимум пересылок с учетом свойств коммуникационной сети не превосходит минимума одинарных пересылок.

Пример 1. 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 принимает 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 , размер блока , элементы массива a размещены (все в одном ПЭ с номером 2), элементы массива b размещены , элементы массива c размещены . 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 , где .

В примере 1 массив a размещен блочно-аффинно $P_a(i) = 2$ = (⌊1/d1⌋ ∗ i + 2) mod p, где d1 – большое число (не меньшее количества итераций цикла), s0 = 2, s1 = 1; определяет блочно-аффинное размещение c , определяет блочно-аффинное размещение c .

Пусть массив a размещен блочно-аффинно. Пусть вхождение a[i + c] требует (не требует) пересылки на итерации , тогда на любой итерации вида , где , это вхождение также требует (не требует) пересылки (здесь d – размер блока).

Теорема. Пусть количество процессорных элементов , где — константы в индексных выражениях цикла, приведенного в листинге 2, тогда минимальное количество одинарных пересылок достигается при размере блока .

Минимизация циклических пересылок для размещений массивов с блоками размера 1 рассматривалась 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 потребуется 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 2. Распределение итераций цикла между ПЭ можно описать как 2 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 (j = 0; j < N; j = j + 1) {
  Y[j] = (X[j - 1] + X[j] + X[j + 1]) / 3;
}
```

Для параллельного выполнения этот цикл преобразуется к виду:

```
for (j2 = 0; j2 < 11/p; j2++) {
    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 рассылки данных broadcast.

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.

Таким образом, данная статья представляет собой шаг на пути к созданию оптимизирующих распараллеливающих компиляторов на высокопроизводительные системы на кристалле нового поколения типа «суперкомпьютер на кристалле».

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

Литература

1. S. G. Ammaev, L. R. Gervich, B. Y. Steinberg. “Combining parallelization with overlaps and optimization of cache memory usage”, PaCT 2017:

Parallel Computing Technologies, Lecture Notes in Computer Science, vol. 10421, pp. 257-264.
2. Z. Gong, Z. Chen, Z. Szaday, D. Wong, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau // An empirical study of the effect of source-level loop transformations on compiler stability / Proceedings of the ACM on Programming Languages. — 11.2018. — С. 1-29.
3. Л.Р. Гервич, Е.Н. Кравченко, Б.Я. Штейнберг, М.В. Юрушкин Автоматизация распараллеливания программ с блочным размещением данных, Сибирский журнал вычислительной математики, т. 18, №1 (2015), с. 41-53.
4. U. Bondhugula. Automatic distributed-memory parallelization and codegeneration using the polyhedral framework, Technical report, ISc-CSA-TR-2011-3, 2011, 10 pp.
5. DVM-система разработки параллельных программ | DVM-система. — URL: http://dvm-system.org/ru/about/ (дата обр. 26.03.2022).
6. В.В. Корнеев Параллельное программирование// Программная инженерия. 2022, — т. 13, № 1, с. 3-16.
7. N.M. Krivosheev, B.Ya. Steinberg / 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.
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. pp. 152-157. — DOI: 10.1109/ISPAN.1999. 778932.
9. Olofsson A. 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 о бр. 26.03.2022).
10. SoC Esperanto. — URL: https://www.esperanto.ai/technology/ (дата обр. 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. И.В. Прангишвили, С.Я. Виленкин, И.Л. Медведев. Параллельные вычислительные системы с общим управлением. — М.: Энергоатомиздат, 1983, 312 с.
13. Процессор НТЦ "Модуль": https://www.cnews.ru/news/top/2019-03-06_svet_uvidel_moshchnejshij_rossijskij_nejroprotsessor (дата обр. 26.03.2022).
14. Штейнберг Б.Я. Блочно-аффинные размещения данных в параллельной памяти. «Информационные технологии», М.: из-во «Новые технологии», №6, 2010 г., 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. (2021). Precompiler for the ACELAN-COMPOS Package Solvers. In: Malyshkin, V. (eds) Parallel Computing Technologies. PaCT 2021. Lecture Notes in Computer Science(), vol 12942, pp. 103-116. Springer, Cham. https://doi.org/10.1007/978-3-030-86359-3_8

References

1. S. G. Ammaev, L. R. Gervich, B. Y. Steinberg. “Combining parallelization with overlaps and optimization of cache memory usage”, PaCT 2017: Parallel Computing Technologies, Lecture Notes in Computer Science, vol. 10421, pp. 257-264.
2. Z. Gong, Z. Chen, Z. Szaday, D. Wong, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau // An empirical study of the effect of source-level loop transformations on compiler stability / Proceedings of the ACM on Programming Languages. — 11. 2018, pp. 1-29.
3. L. R. Gervich, E. N. Kravchenko, B. Y. Steinberg, M. V. Yurushkin. “Automatic program parallelization with block data distribution”, Num. Anal. Appl., 8:1 (2015), pp. 35–45
4. U. Bondhugula. Automatic distributed-memory parallelization and codegeneration using the polyhedral framework, Technical report, ISc-CSA-TR-2011-3, 2011, 10 pp.
5. DVM-sistema razrabotki parallel'nyh programm DVM-sistema. — URL: http://dvm-system.org/ru/about/ (Accessed 26.03.2022).
6. V.V. Korneev Parallel'noe programmirovanie //Programmnaya Ingeneria. 2022, v. 13, № 1, pp. 3-16.
7. N.M. Krivosheev, B.Ya. Steinberg / 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.
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. — pp. 152-157. — DOI: 10.1109/ISPAN.1999. 778932.
9. Olofsson A. 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/ (Accessed 26.03.2022).
10. SoC Esperanto. — URL:https://www.esperanto.ai/technology/ (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. I.V. Prangishvili, S.YA. Vilenkin, I.L. Medvedev. Parallel'nye vychislitel'nye sistemy s obshchim upravleniem. — M.: Energoatomizdat, 1983, 312 s.
13. Processor NTC "Modul'": https://www.cnews.ru/news/top/2019-03-06_svet_uvidel_moshchnejshij_rossijskij_nejroprotsessor (Accessed 26.03.2022).
14. Shtejnberg B.Ya. Blochno-affinnye razmeshcheniya dannyh v parallel'noj pamyati. «Informacionnye tekhnologii», M.: iz-vo «Novye tekhnologii», №6, 2010 g., 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. (2021). Precompiler for the ACELAN-COMPOS Package Solvers. In: Malyshkin, V. (eds) Parallel Computing Technologies. PaCT 2021. Lecture Notes in Computer Science(), vol 12942, pp. 103-116. Springer, Cham. https://doi.org/10.1007/978-3-030-86359-3_8