УДК: 004.272.2, 004.051, 519.685.1, 519.686.2, 519.687.1

ПРОБЛЕМЫ ЭФФЕКТИВНОСТИ АВТОМАТИЧЕСКОГО ДИНАМИЧЕСКОГО РАСПАРАЛЛЕЛИВАНИЯ ВЫЧИСЛЕНИЙ ДЛЯ МНОГОПРОЦЕССОРНЫХ КОМПЬЮТЕРНЫХ СИСТЕМ СО СЛАБОЙ СВЯЗЬЮ

Р.И. Левченко, А.А. Судаков, С.Д. Погорелый, Ю.В. Бойко

Киевский национальный университет имени Тараса Шевченко.
01033, Киев, ул. Владимирская, 64,
факс. 380 (44) 526-1214, тел. (44) 259 0247,
E-mail: rmn01@mail.ru

Рассматривается проблема автоматического динамического распараллеливания вычислений для много-процессорных компьютерных систем со слабой связью. В основе исследований используется авторская система автоматического распараллеливания вычислений DDCI. В работе исследуется проблема эффективности динамического распараллеливания вычислений, анализируются подходы к правильному написанию программ, подлежащих динамическому распараллеливанию. Выдвигаются требования, к динамически распараллеливаемым программам, позволяющие добиться максимальной эффективности распараллеливания.

The automatic dynamic parallelizing problem of calculations for multiprocessing computer systems with loose connection is considered. At the inherently of researches we use the author's system of automatic dynamic parallelizing (DDCI). In work the problem of efficiency of calculations parallelizing is investigated, approaches to a correct creation of parallel dynamic programs are analyzed. Demands, to dynamically parallelizing of programs are made, for calculations efficiency optimization.

Введение

Автоматическое распараллеливание вычислений является актуальной и полностью не решенной проблемой в науке. Параллельные вычисления крайне важны в отраслях науки использующих машинное моделирование [1]. В связи с постоянным ростом и развитием многопроцессорных компьютерных систем актуальной становится проблема эффективного использования многопроцессорных компьютерных систем со слабой связью [2]. Самые популярные на сегодня подходы для написания параллельных программ, как правило, лишены автоматизации процесса распараллеливания [3, 4], либо не эффективны для многопроцессорных компьютерных систем со слабой связью [5].

Сегодня существует целый ряд нерешенных проблем, связанных со статическим распараллеливанием вычислений для многопроцессорных компьютерных систем со слабой связью [6, 7]. Основные проблемы связанны с невозможностью эффективного статического распараллеливания последовательных программ, не зная их входных данных [8]. В связи с этим активно развивается направление автоматического динамического распараллеливания вычислений [9, 10, 11]. Направление динамического распараллеливания вычислений лишено проблем, связанных с неизвестностью входных данных, так как распараллеливание проводится на этапе выполнения программы, когда все параметры уже заданны. Динамическое распараллеливание при этом тоже не лишено серьезных проблем, одна из основных – это проблема эффективного распределения вычислений в реальном времени [12, 13]. Первопричина этой проблемы связана с неправильным выбором степени блочности последовательного алгоритма. Решение данной проблемы – цель статьи.

Исследуется проблема эффективного разбиения последовательного алгоритма на блоки для возможности последующего динамического распараллеливания. Исследование проводится на примере распараллеливания прямого хода решения СЛАУ методом Гаусса. В качестве системы автоматического динамического распараллеливания вычислений используется авторское решение – система DDCI [14]. Эффективность и уместность системы DDCI при динамическом распараллеливании уже обсуждались в предыдущих работах [15].

Введение в основные принципы DDCI системы

Система DDCI (Dynamic Distribution Calculations Interface) разработана на основе построения и анализа динамически меняющихся графов последовательных программ. Особенность такого подхода состоит в то, что система динамического распараллеливания вычислений (DDCI) видит алгоритм последовательной программы как граф, состоящий из помеченных чистых функций (вершины графа) и потоков данных между ними (ребра графа).

Чистые функции – это функции, которые однозначно вычисляются по заданным параметрам. Чистые функции не должны хранить в себе или где бы то ни было еще данные между двумя вызовами. Такие функции должны работать только с теми данными, которые им были переданы в качестве параметров, и возвращать результаты своей работы также через эти параметры. Они не могут использовать сторонние методы передачи данных (например, ввод с клавиатуры и вывод на консоль), так как это противоречит принципу о не хранении данных за пределами функции.

Помеченная чистая функция – это функция, которая была помечена программистом как подлежащая распараллеливанию. Для DDCI системы пометка чистых функций производится с помощью макросных деклараций следующего вида (исходные код приведен на языке C++):

Пример 1.

```
double Integrate( double Xmin, double Xmax, double Xstep ){
   double Sum = 0;
   for(double X = Xmin; X < Xmax; X += Xstep)
      Sum += sin(X);
   return Sum;
}
DDCI_FUNCTION( /*return*/ double, /*function name*/ Integrate,
   /*Xmin*/ READ, double, /*Xmax*/ READ, double, /*Xstep*/ READ, double );
```

Пример 2.

```
void MatrixSum( float * A, float * B, long BlockSize ){
    // A и B это двумерные массивы размерностью BlockSize* BlockSize
    long i_max = BlockSize * BlockSize;
    for(long i=0;i<i_max;i++)
        A[i] += B[i];
}
DDCI_FUNCTION( /*return*/ void, /*function name*/ MatrixSum,
    /*A*/ MODIFY, float*, /*B*/ READ, float*, /*BlockSize*/ READ, long );
```

В примере 1 приведено объявление функции интегрирования и пометка ее как чистой. В примере 2 показано объявление функции суммирования для двух массивов. Из примеров видно, что при объявлении функций `Integrate`, `MatrixSum` и их пометке макросом `DDCI_FUNCTION`, у чистых функций декларируются все параметры, которые используются функцией для ее работы, и каждый из параметров помечается одним из трех режимов:

`READ` – параметр используется для передачи данных в функцию, и при выполнении функции не модифицируется;

`WRITE` – параметр используется для передачи результатов вычисления данной функции в другие участки программного кода;

`MODIFY` – параметр сперва используется для передачи данных в функцию, но при выполнении функции модифицируется и в дальнейшем используется для передачи результатов вычисления данной функции в другие участки программного кода.

Указание режима, в котором используются параметры, необходимо для правильного построения направленного графа последовательного алгоритма программы.

Поток данных – это блок данных известного размера, который является результатом выполнения одной чистой функции и используется как исходные данные для выполнения следующих чистых функций.

В результате распараллеливание последовательного алгоритма происходит только за счет одновременного выполнения нескольких чистых функций, независящих друг от друга по данным. Эффективность распараллеливания будет зависеть напрямую от правильности разбиения последовательного алгоритма на чистые функции.

Основные отличия последовательных алгоритмов от параллельных аналогов

Главным отличием последовательных и параллельных алгоритмов является блочность. Эта характеристика описывает, как качественно алгоритм разбит на чистые функции. Для понимания важности этой характеристики достаточно сказать, что подавляющее большинство параллельных алгоритмов, которые реализуются для систем со слабой связью, построены на основе явных и неявных чистых функций [16].

Чистые функции являются фундаментом для параллельного программирования на системах с распределенной памятью. Причина этого в том, что такие функции всегда четко декларируют данные, с которыми они будут работать, а значит всегда известно, какие данные и откуда надо собрать на одной вычислительной станции для выполнения конкретной чистой функции. Можно заметить, что если взять алгоритм программы и представить его в виде чистых функций (вершины графа), а потом связать вершины графа с помощью потоков данных между их функциями (ребра графа), то получится граф алгоритма описывающий не полностью весь алгоритм, а только связи между основными блоками алгоритма. Так как в системах со слабой связью передача данных между потоками является длительной операцией, то представление алгоритмов в виде графа потоков данных между вычислительными блоками является идеальным. Приведен пример, поясняющий, как можно представить алгоритм в виде потоков данных между чистыми функциями.

Пример 3.

```
double algorithm( double a, double b){
   double x = cos( a );
   double y = sin( b );
   return sin( y ) + x
}
```

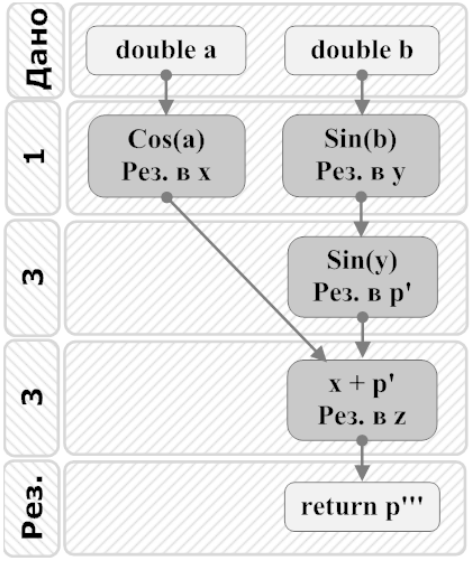

Рис. 1. Блок-схема для примера 3

Из приведенного примера (рис. 1) видно что большая часть алгоритма последовательна, но функции `cos( a )` и `sin( b )` можно вычислять параллельно. То, что последовательные и параллельные части алгоритма легко видны на таких графах и поясняет, почему их удобно использовать при написании параллельных программ для многопроцессорных компьютерных систем со слабой связью.

Теперь рассмотрим важнейшую характеристику всех чистых функций, которая определяет, будет ли параллельный алгоритм работать быстрее последовательного или нет. Этой характеристикой является отношение времени работы функции к времени затрачиваемого на передачу всех необходимых данных по сети для ее вычисления. В дальнейшем данная характеристика будет называться эффективностью чистой функции.

Фактически эффективность любой чистой функции прямо пропорциональна сложности алгоритма в этой функции. Получается что, чем ниже сложность алгоритмов в чистых функциях, тем более жесткие требования накладываются на скорость передачи данных между потоками. В результате, если рассматривать функции со сложностью, близкой к линейной, то можно говорить о невозможности их эффективного распараллеливания, так как время передачи данных всегда будет не меньше времени выполнения функции.

Теперь подведем итоги. Если последовательный алгоритм реализован на основе чистых функций, у которых время выполнения алгоритма больше времени передачи их данных между процессорами, то алгоритм потенциально может быть распараллелен. Эффективность распараллеливания будет зависеть от эффективности чистых функций и их графа связанности по данным.

Проблема эффективности чистых функций

Для динамического распараллеливания вычислений проблема правильного разбиения последовательного алгоритма на блоки крайне важна. Причина в том, что системы планирования потоков могут скомпенсировать гетерогенность компьютерной системы и учесть меняющиеся характеристики каналов для передачи данных. При этом, если последовательный алгоритм неправильно разбит на блоки, то параллельный алгоритм может работать даже медленнее последовательного аналога.

Неправильное разбиение последовательного алгоритма на блоки связано с двумя ограничениями:

–если граф программы преимущественно состоит из последовательно идущих блоков или количество возможных параллельных блоков слишком мало;

–эффективность чистых функций мала, для того чтобы скомпенсировать потери времени на динамическое распараллеливание программы и передачу данных между процессорами.

Для наглядности рассмотрим эти причины на примере алгоритма прямого хода для решения СЛАУ методом Гаусса. Данный алгоритм можно разбить на чистые функции тремя разными способами, все три примера представлены на рис. 2.

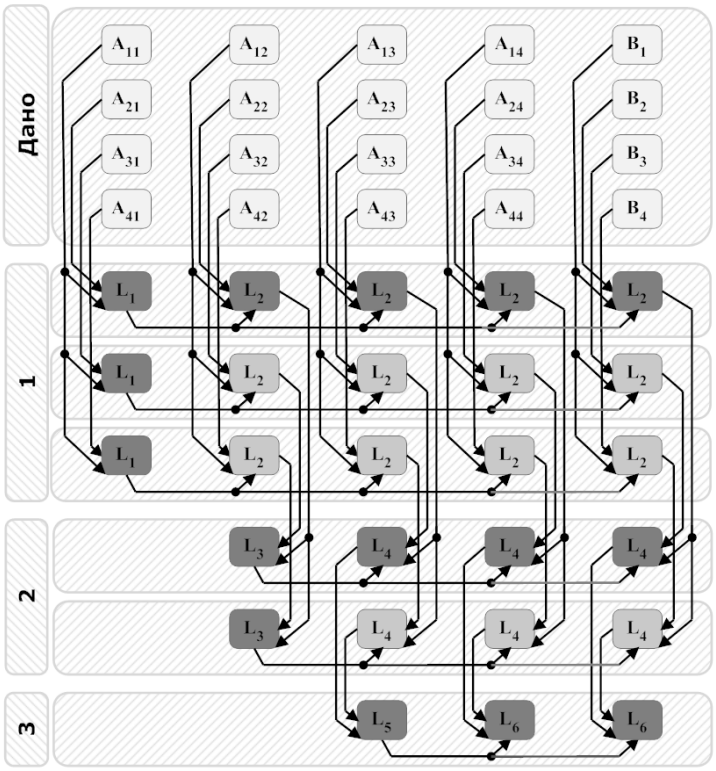

Рис. 2. Блок-схема алгоритма прямого хода для решения СЛАУ, вариант 1

Первый способ – это реализация на основе примитивных блоков, помеченных L*, где L* – операция промежуточного расчета одного элемента в матрице * обозначает этап вычисления параллельного алгоритма программы. Анализ возможных параллельных функция приведен в табл. 1.

Таблица 1. Список чистых функций, для сильного разбиения прямого хода решения СЛАУ (размерность 4)

Имя этапа	Количество параллельных функций	Логика функции
L-1	3	$f = a_{ij} / a_{kj}$
L-2	12	$f = b * a_{ks} - a_{is}$
L-3	2	$f = a_{ij} / a_{kj}$
L-4	6	$f = b * a_{ks} - a_{is}$
L-5	1	$f = a_{ij} / a_{kj}$
L-6	2	$f = b * a_{ks} - a_{is}$

Из табл. 1 видно, что максимального ускорения можно достигнуть на 12-процессорной рабочей станции. Предположим, для упрощения расчетов, что все функции вычисляются одинаковое количество времени. Взяв 12-процессорную рабочую станцию, получаем 6 после-довательных этапов для 26 функций. Время работы параллельного алгоритма составит $6/26 \approx 23\%$ по сравнению с последовательным аналогом.

В результате получается достаточно высокое ускорение, но это без учета потерь на планирование и передачу данных. Если учесть что функции, на которые разбит последовательный алгоритм, достаточно элементарны и отрабатывают быстрее алгоритмов анализа графа, то полу-ченная параллельная реализация будет работать намного медленнее последовательной. Получается, что сам граф алгоритма можно распараллелить эффективно, но из-за неправильного выбора помеченных чистых функций распараллеливание будет не эффективным.

Рассмотрим второй способ распараллеливания алгоритма прямого хода метода Гаусса для решения СЛАУ, а именно распараллеливание по строкам табл. 2.

Таблица 2. Список чистых функций для построчного распараллеливания алгоритма Гаусса (размерность 4)

Имя этапа	Количество параллельных функций	Логика в функциях
I–1	3	Цикл вычитания из значений *i*-й строки значений *j*-й строки с умножением на коэффициент
I–2	2	
I–3	1	

Последовательные этапы второго метода распараллеливания помечены именами I-*, где * обозначает номер этапа параллельного вычисления.

Из примера видна невысокая эффективность распараллеливания для данной задачи. Так, для трех – процессорной системы время работы параллельного алгоритма составит $3/6 \approx 50\%$. При этом, если увеличивать размерность задачи, то эффективность распараллеливания будет расти. Например, для СЛАУ из 6 уравнений (в предыдущем примере их 4) максимальная эффективность распараллеливания составит $5/15 \approx 33\%$. Блок-схема алгоритма прямого хода для решения СЛАУ размерности 6 показана на рис. 3. Описание соответствующих чистых функций их количества и этапов параллельного алгоритма приведены в табл. 3.

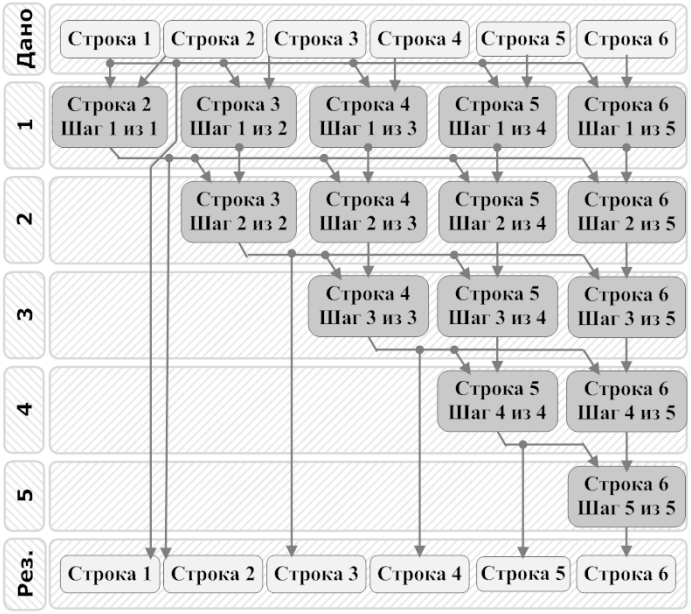

Рис. 3. Блок-схема алгоритма прямого хода метода Гаусса для решения СЛАУ, построчное распараллеливание (размерность 6)

Таблица 3. Список чистых функций для построчного распараллеливания (размерность 6)

Имя этапа	Количество параллельных функций	Логика в функциях
1	5	Цикл вычитания из значений *i*-й строки значений *j*-й строки, с коэффициентом
2	4	
3	3	
4	2	
5	1	

Так как теперь чистые функции состоят не из простейших математических операций как в первом примере, а имеют в себе одномерный цикл, то предложенный способ распараллеливания будет эффективным для вычислений на многопроцессорных компьютерных системах с общей памятью. Использовать этот подход для многопроцессорных систем со слабой связью по-прежнему невозможно из-за линейной сложности алгоритма в чистых функциях.

Третий способ распараллеливания алгоритма прямого хода для решения СЛАУ методом Гаусса – это блочное распараллеливание. Под блоком подразумевается объединение нескольких подряд идущих строк из СЛАУ в один блок. У такого способа распараллеливания граф (рис. 3) и основные характеристики аналогичны способу построчного распараллеливания вычислений. Разница будет следующая:

– уменьшение количества помеченных чистых функций в графе программы, из-за объединения нескольких чистых функций, из второго алгоритма, в одну. При слишком сильном уменьшении количества чистых функций это будет приводить к понижению эффективности распараллеливания;

– повышение сложности алгоритма чистых функций от линейной до квадратичной. Это позволит использовать данный алгоритм для систем со слабой связью. Эффективность чистых функций будет расти прямо пропорционально количеству строк из СЛАУ в одном блоке.

Из вышеприведенных трех способов распараллеливания видно, что при использовании динамического распараллеливания вычислений на программиста ложится сложная задача разбиения последовательного алгоритма на чистые функции. Эффективность чистых функций и их количество в алгоритме определяют эффективность всего процесса распараллеливания. При создании параллельных алгоритмов с низкими требованиями к каналам связи приходится разбивать последовательный алгоритм на большое количество чистых функций высокой сложности, а это взаимопротиворечащие требования, та как при повышении сложности помеченных чистых функций резко начинает падать их количество в графе параллельной программы.

Экспериментальное исследование зависимости качества динамического распараллеливания алгоритма от эффективности используемых чистых функций

Полное тестирование зависимости эффективности распараллеливания от размера и формы графа программы, а также от эффективности помеченных чистых функций является достаточно сложной и емкой задачей. Поэтому в данной статье будут приведены только основные следствия экспериментального тестирования, позволяющие выделить только самые основные зависимости эффективности динамического распараллеливания вычислений. Полное описание эксперимен-тальных результатов по оптимизации степени блочности последовательных алгоритмов для эффективного их динамического распараллеливания будут приведены в отдельной работе.

Для написания параллельных программ для многопроцессорных компьютерных систем на основе межпроцессорного транспорта Gigabit Ethernet желательно использовать чистые функции с эффективностью около 10, чтобы добиться оптимального распараллеливания алгоритма. Функции с эффективностью ниже 5 являются крайне нежелательными, и их частое использование приводит к резкому падению общей эффективности распараллеливания всей задачи.

Эффективность функции – это отношение теоретического времени вычисления функции к минимальному времени передачи данных между процессорами для ее вычисления. Теоретического времени вычисления функции соответствует времени, затрачиваемому функцией на выполнение при идеально эффективном распараллеливании. Минимальное время передачи данных для запуска функции вычисляется как отношение объема данных, которые необходимо передать между процессорами для запуска функции, к пиковой производительности каналов передачи данных между процессорами.

Еще одним экспериментальным следствием является требование к графу программы. Граф параллельной программы должен потенциально распараллеливаться для количества потоков в 3–5 раз больше, чем реально используется в программе. Несоблюдение этого условия приводит к резкому понижение эффективности системы балансировки нагрузки между процессорами, и как следствие – снижению общей эффективности распараллеливания для конкретной задачи.

Выводы

В результате вышеизложенного можно сдеелать следующие выводы. Для того чтобы добиться эффективного распараллеливания последовательного алгоритма с помощью системы динамического распределения вычислений, нужно придерживаться следующих правил.

- Все основные вычисления программы должны находиться в помеченных чистых функциях или вызываться из них.
- Основная часть времени работы программы должна проходить в пределах чистых функций, время работы которых значительно больше времени передачи данных между потоками плюс время работы системы планирования потоков. Это означает, как правило, такие функции должны иметь в себе как минимум двухуровневый цикл, который позволит скомпенсировать все потери времени, связанные с планированием и запуском чистых функций.

- Граф последовательной программы должен иметь минимум последовательных участков, иначе при динамическом анализе графа может не найтись эффективного способа распараллеливания задачи на заданное число потоков.
- В программе не должно быть излишка чистых функций, так как это может привести к существенному замедлению системы планирования.

1. *Chervenak A., Foster I., Kesselman C., Salisbury C., Tuecke S.* The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets // J. of Network and Computer Applications, 23:187–200, 2001 (based on conference publication from Proceedings of NetStore Conference 1999).
2. *Gropp W., Lusk E., Sterling T.* Beowulf Cluster Computing with Linux, 2nd Edition. – MIT Press, 2003. – 504 p.
3. *Geist G., Kohl J., Manchel R. and Papadopoulos P.* New Features of PVM 3.4 and Beyond // PVM Euro Users' Group Meeting, September, 1995, Lyon, France, Hermes Publishing, Paris. – P 1–10.
4. *Jeffrey M. Squyres, Bill Saphir, and Andrew Lumsdaine.* The Design and Evolution of the MPI-2 C++ Interface. In Proceedings, 1997 // Intern. Conf. on Scientific Computing in Object-Oriented Parallel Computing, Lecture Notes in Computer Science, Springer-Verlag, 1997.
5. *Chapman B., Jost G., R. van der Pas, D.J. Kuck* (foreword), Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press (October 31, 2007).
6. High Performance Fortran. Language Specification. Version 1.0, Jan.25, 1993.
7. *Rui Yang, Jie Cai, Alistair P. Rendell, V. Ganesh.* Use of Cluster OpenMP with the Gaussian Quantum Chemistry Code: A Preliminary Performance Analysis // Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism. 2009 – 5568. – P. 53 – 62.
8. *Levchenko R.I., Sudakov O.O., Pogorelij S.D, Bojko Y.V.* A System of Automatic Dynamic Paralleling of Computations for Multiprocessor Computer Systems with Weak Connection (DDCI) // USiM. – 2008. – N 3. – P. 66–72.
9. *Abramov S., Adamovitch A., Kovalenko M.* T-system: programming environment providing automatic dynamic parallelizing on IP-network of Unix-computers // Report on 4-th International Russian-Indian seminar and exibition, Sept. 15–25, 1997, Moscow.
10. *N. Alexey.* Salnikov PARUS: A Parallel Programming Framework for Heterogeneous Multiprocessor Systems // Lecture Notes in Computer Science. Recent Advantages in Parallel Virtual Machine and Message Passing Interface. – 2006. –4192. – P. 408409.
11. *Levchenko R.I., Sudakov O.O.* DDCI: Dynamic parallelizing system for high performance computing clusters // Proceedings of the seventh international young scientists' conference on applied physics. – P. 147-–148.
12. Rolf Rabenseifner, Georg Hager, Gabriele Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," // Parallel, Distributed and Network-based Processing, 2009. pp.427-436
13. *Levchenko R.I., Sudakov O.O., Maistrenko Yu.L.* Parallel software for modeling complex dynamics of large neuronal networks // Proc. 17th International Workshop on Nonlinear Dynamics of Electronic Systems, Rapperswil, Switzerland. June 21–24, 2009. – P. 34–37.
14. *Levchenko R.I., Sudakov O.O., Pogorilyy S.D.* DDCI: Interface for transparent parallelizing of calculations. // Proceedings of the ninth international young scientists' conference on applied physics. – P. 112.
15. *Levchenko R.I., Sudakov O.O., Pogorelij S.D.* DDCI: Simple Dynamic Semiautomatic Parallelizing for Heterogeneous Multicomputer Systems // IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications. 21–23 September 2009, Rende (Cosenza), Italy. – P. 70–74.
16. *Keren A., Barak A.* Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster. // IEEE Tran. Parallel and Distributed Systems 1 (14), (2003). – P. 39–50.