УДК 004.27; 004.25; 004.382.2

Е.В. ЕЛИСЕЕВА

О РАСПАРАЛЛЕЛИВАНИИ ПОЛЬЗОВАТЕЛЬСКИХ ЗАДАЧ В РАСПРЕДЕЛЕННЫХ КОМПЬЮТЕРНЫХ СИСТЕМАХ ТИПА “ПРОЦЕССОР-В-ПАМЯТИ”

Анотація. *Наведено результати аналізу існуючих підходів і алгоритмів розділення програм користувача для їх паралельної реалізації на PIM-системі. На основі сукупності висновків щодо їх складності і високої трудомісткості сформульовані основні положення концепції побудови нового, простішого і менш трудомісткого алгоритму, приймаючи за основу особливості архітектурно-структурної організації PIM-систем.*
Ключові слова: *алгоритм розпаралелювання програми користувача, PIM-системи.*

Аннотация. *Приведены результаты анализа существующих подходов и алгоритмов разделения пользовательских программ для их параллельной реализации на PIM-системе. На основе совокупности выводов по их сложности и высокой трудоемкости сформулированы основные положения концепции построения нового, более простого и менее трудоемкого алгоритма, принимая за основу особенности архитектурно-структурной организации PIM-систем.*
Ключевые слова: *алгоритм распараллеливания пользовательской программы, PIM-системы.*

Abstract. *Results of the analysis of existing approaches and algorithms of division of the userland programs for their parallel realization on PIM-system are resulted. On the basis of set of conclusions on their complexity and high labour input the substantive provisions of the concept of construction of new, more simple, and less labour-consuming algorithm are formulated, assuming as a basis features of the architecturally-structural organisation of PIM-systems.*
Key words: *algorithm of parallelizing of the userland program, PIM-systems.*

1. Введение

Достижение высокой производительности компьютерных систем (КС) существенно зависит от решения так называемых “служебных процедур” распределения памяти, размещения данных и особенно от распараллеливания алгоритма решаемой задачи пользователя. Необходимость оптимального решения этих процедур особенно актуальна для современных и перспективных распределенных высокопроизводительных КС, поскольку их архитектурно-структурная организация как раз и исходит из потребности их реализации. Наиболее трудоемкой и в то же время наиболее значимой с точки зрения производительности является процедура распараллеливания пользовательской задачи, что привело к огромному количеству исследований и публикаций, освещающих решение этой задачи применительно к кластерным и GRID-системам. Однако количество подобного рода публикаций применительно к системам типа “Processor-in-Memory” или PIM-системам весьма ограниченно, и тем более их комплексный анализ в отечественной и зарубежной литературе отсутствует. К тому же каждый подход к решению данной задачи, как правило, основан на допущениях, влияние риска принятия которых на конечный результат загрузки процессоров и тем самым на производительность системы в целом не оценивается. В соответствии с этим в данной статье приведен краткий анализ методов распараллеливания приложений для PIM-систем, определяются их допущения и в ряде случаев замечания (недостатки с точки зрения автора статьи), для доказательства которых приведен более подробный анализ одного из типовых подходов к решению задачи распараллеливания приложений, на основе чего предлагаются основные принципы распределения приложений для КС этого класса, основываясь на особенностях их архитектурно-структурной организации.

2. Краткий анализ существующих подходов к распределению приложений для PIM-систем

В работах [1–11] описываются автоматизированная система типа SAGE (Statement-Analysis-Grouping-Evaluation) и автоматизированная система Octans, которые представляют собой наборы программ, реализующих разделение и планирование пользовательской программы для ее выполнения на PIM-системе с учетом особенностей P.Host (хост-процессора) и P.Mem (процессора памяти). При этом в основе подхода, принятого в [1–4], используется упрощенная модель PIM-системы, которая содержит один P.Host и один P.Mem, а в [10–11] рассматривается модель PIM-системы, состоящая из одного P.Host и четырех P.Mem. В [5, 7, 8] приводятся модули SAGE, ориентированные на PIM-систему с одним хост-процессором и n процессорами памяти.

Основные стадии работы системы SAGE при разделении программы пользователя кратко описаны в [1–3], начиная от операторного разбиения исходной программы до формирования плана её выполнения на хост-процессоре и соответственно на процессоре памяти. При этом основной единицей анализа программы является оператор в цикле (используется операторный метод), а не итерация. В [4] предлагается дополнить систему SAGE механизмом new low-power transformation mechanism (новый низкозатратный механизм трансформации), который назван POERS (Performance-Oriented Energy Reduction Scheduling) и призван уменьшить потребление энергии для PIM-системы без потери производительности. При этом усложняется процедура оценки веса каждого блока, вес оценивается уже не двумя значениями: вес для хост-процессора и вес для процессора памяти, а четырьмя значениями: вес задержки (delay weights) и энергетический вес (energy weight) отдельно для хост-процессора и для процессора памяти. В [5] рассмотрены два механизма планирования для PIM-системы: первый назван 1H-1M (one-host and one-memory processors) – механизм планирования для упрощенной PIM-системы, включающей один хост-процессор и один процессор памяти. Второй механизм определен как 1H-nM (one-host and n-memory processors) – механизм планирования для PIM-системы, содержащей один хост-процессор и n процессоров памяти. В [6] рассмотрен генератор кода для SAGE, который преобразовывает исходную программу для PIM-системы согласно графу гиперблоков (HBG) и плану выполнения, полученных в результате работы других модулей SAGE. При этом генератор кода создает подпрограммы, вставляет необходимые аргументы, определяет места вызываемых и вызывающих процедур и генерирует программу для выполнения на PIM-системе. В [7] приведен механизм оценки веса в виде пары значений: веса для хост-процессора и веса для процессора памяти, а также механизм планирования 1H-nM, а в [8] рассмотрены проблемы балансировки загрузки при планировании задач с помощью системы SAGE. В [9] предложен механизм разбиения программы для выполнения на PIM-системе, который включает операторное разбиение для идеальных циклов и модуль конструирования гиперблоков. И, наконец, в [10, 11] описана система Octans, которая, как и SAGE, разбивает первоначальную программу на блоки операторов, производит соответствующий граф зависимости блоков, генерирует план выполнения и генерирует соответствующие потоки для хост-процессора и процессоров памяти. При планировании блоков программы для различных процессоров используют метод критического пути (МКП).

Анализ этих работ показал, что процесс распределения приложений для PIM-систем содержит длинную цепочку различных процедур с огромным количеством переборов, вследствие чего такой процесс является очень трудоемким и занимает большое количество времени. Доказательством тому могут служить рис. 1, 2, где приведены цепочки реализации одного из известных подходов к распределению приложений для PIM-систем, используя упрощенную модель PIM-системы с одним P.Host и несколькими P.Mem [9–11].

3. Алгоритм распараллеливания пользовательской программы согласно рис. 1 и 2

На рис. 1 и 2 показано четыре крупных (основных) модуля**:** модуль операторного разбиения, построения графа гиперблоков, модуль оценки веса блоков и модуль возвратного механизма планирования.

Описание алгоритма

В работах [1–3, 10–11] модуль 2 исключен и рассматривается распараллеливание только для операторов идеально вложенных циклов (модуль 1), а в [6–9] рассматривается как модуль операторного разбиения, так и модуль построения гиперблоков. Рассмотрим несколько подробнее каждый из этих модулей, введя при этом необходимые понятия.

Модуль 1. Операторное разбиение

Граф алгоритма – это граф, который описывает информационные зависимости алгоритма. Вершины такого графа соответствуют отдельным операторам алгоритма, а ребра (дуги) указывают на наличие зависимости между вершинами (операторами), которые они соединяют. Такой граф является ориентированным (все ребра направлены).

Пусть S_k и S_h – операторы последовательной программы, причем S_k расположен перед S_h. Между S_h и S_k можно выделить два типа зависимостей: зависимости по данным и зависимости управления. В свою очередь, зависимости по данным также могут быть различных типов [12, 13]. В табл. 1 рассмотрены типы зависимостей между операторами.

Таблица 1. Зависимости между операторами алгоритма

Тип зависимости между операторами		Содержание: Оператор S_h имеет зависимость, указанную в первом столбце таблицы, от оператора S_k, если	Пример	Пояснения к примеру
Зависимости по данным	Истинная зависимость	S_h считывает из ячейки, в которую записывает S_k	S_1 : A = 3 S_2 : B = A S_3 : C = B	S_3 зависит от S_2, S_2 зависит от S_1, S_3 зависит от S_1. Порядок следования этих операторов не может быть изменен, они не могут выполняться параллельно
	Антизависимость	S_h записывает в ячейку, из которой S_k считывает	S_1 : B = 3 S_2 : A = B+1 S_3 : B=7	Между S_3 и S_2 есть антизависимость, поскольку S_3 записывает значение B, которое используется S_2. Эти операторы параллельно выполняться не могут. Переименование переменных может устранить эту зависимость
	Зависимость по выходу	S_h записывает данные в ту же ячейку памяти, что и S_k	S_1 : A= 2 * X S_2 : B = A/ 3 S_3 : A= 9 * Y	Между S_3 и S_1 есть зависимость вывода, изменение порядка команд изменит конечное значение. Эти команды не могут быть выполнены параллельно. Эту зависимость можно устранить переименованием переменных
Управляющая зависимость		S_k определяет, должен оператор S_h выполниться или нет	S_1 : if x>y then goto M1 S_2 : a = 2 * x S_3 : M1: ….	Типичный пример: зависимость управления между условием и операторами в соответствующих истинных/ложных телах. Оператор S_2 имеет зависимость управления от оператора S_1

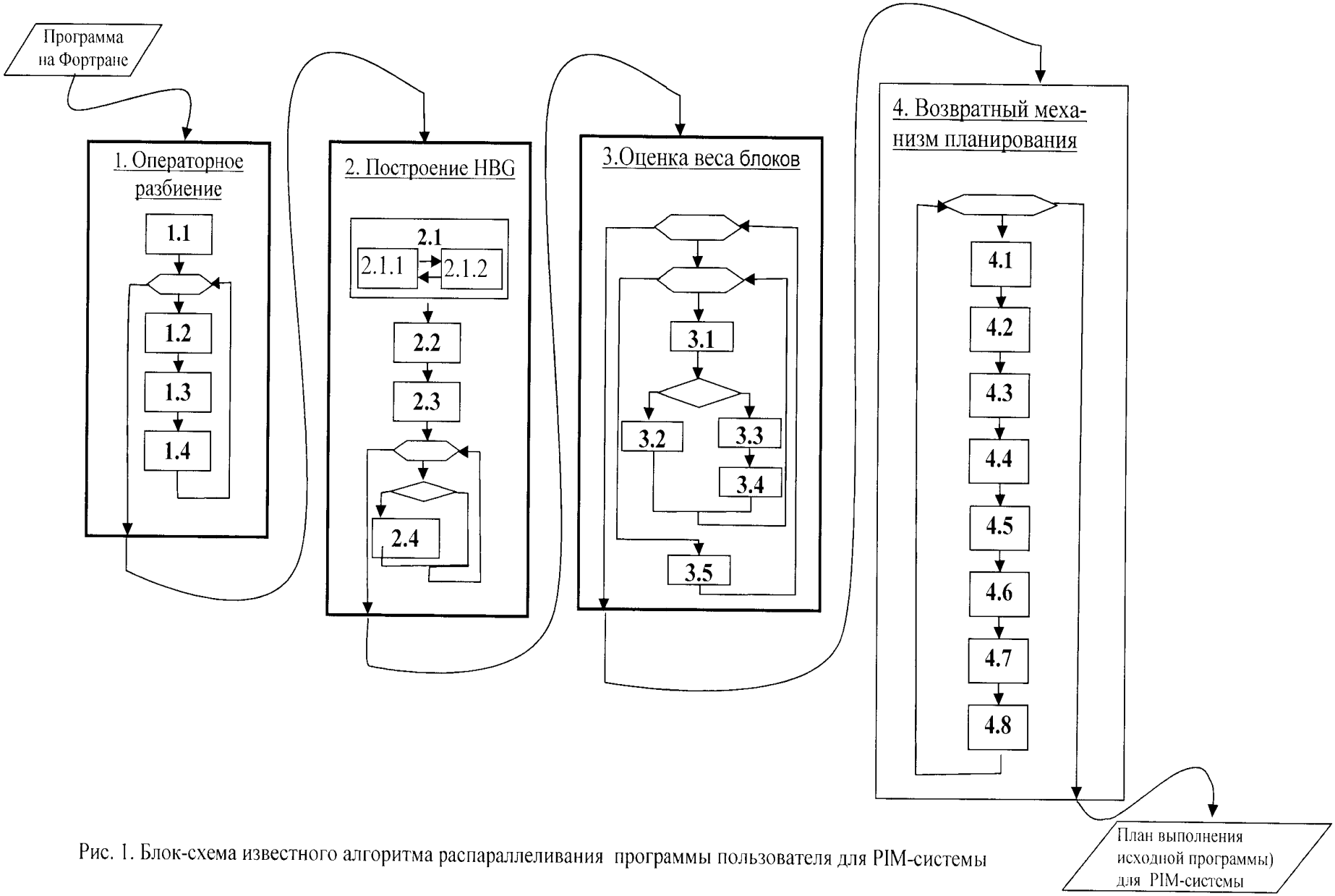
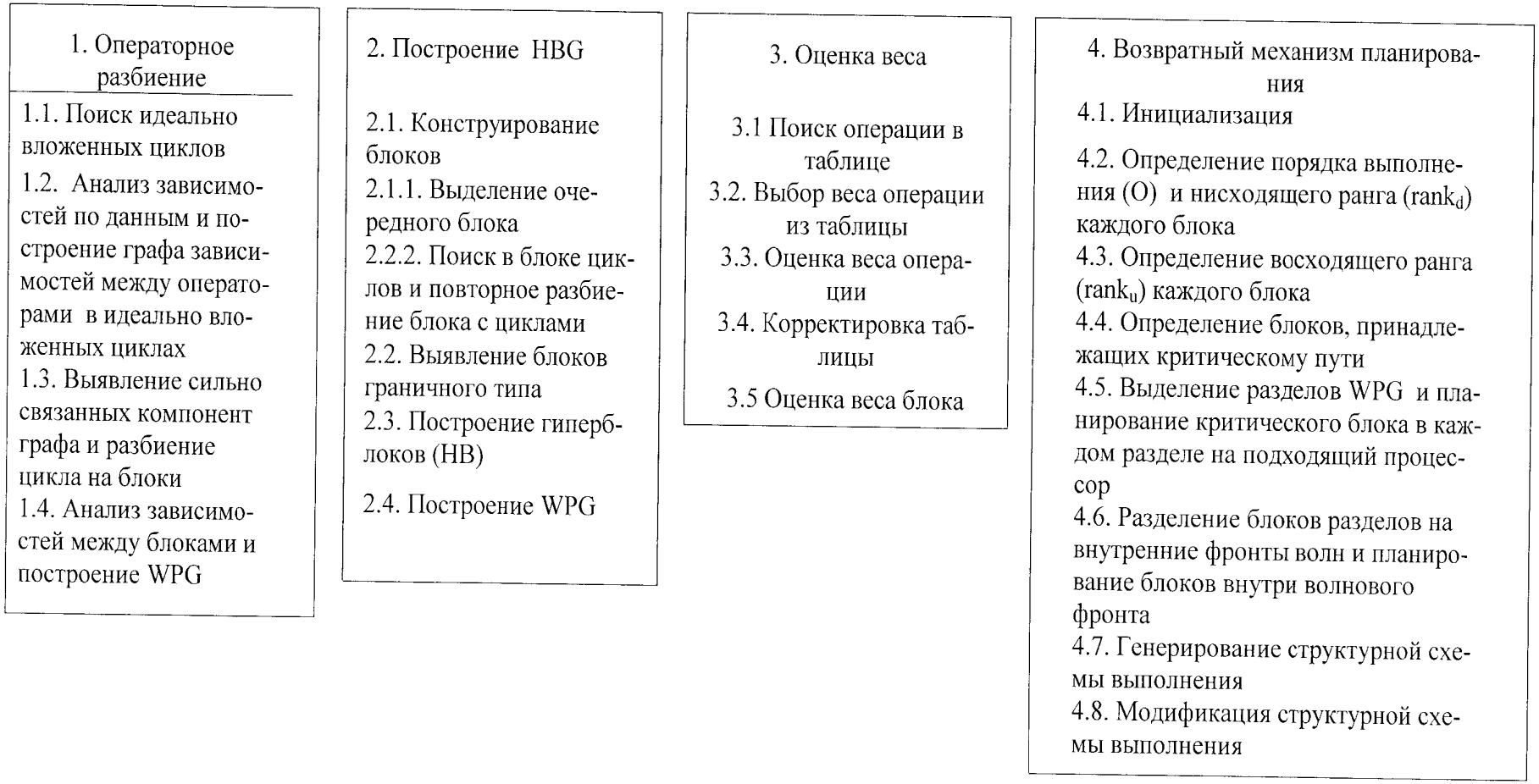


Рис. 1. Блок-схема известного алгоритма распараллеливания программы пользователя для PIM-системы



Используемые сокращения:

WPG (Weighted Partition Dependence Graph) – взвешенный граф зависимости разбиения.
HBG (Hyper Block Graph) – граф гиперблоков.
HB (Hyper Block) – гиперблок.

Рис. 2. Содержательная сущность каждого модуля, приведенного на рис. 1

Граф алгоритма может быть детерминированный (между операторами нет зависимостей управления) и недетерминированный (в противоположном случае).

Сильно связанная компонента в ориентированном графе – это его подграф, любые две вершины которого являются сильно связанными вершинами данного ориентированного графа. Две вершины s и t любого графа сильно связаны, если существует ориентированный путь из s в t и ориентированный путь из t в s. Любая вершина орграфа считается сильно связанной сама с собой.

Обозначим цикл как $L=(I_1, I_2, \ldots, I_n)(S_1, S_2, \ldots, S_d)$, где I_j $(1 \le j \le n)$ – индекс цикла, и S_k $(1 \le k \le d)$ – оператор тела цикла, который может быть оператором присваивания или другим циклом. Для данного цикла L на графе зависимости G определим разбиение на блоки Π для множества операторов $\{S_1, S_2, \ldots, S_d\}$ таким способом, что S_k и S_l, $1 \le k \le d$, $1 \le l \le d$, $k \ne \lambda$ находятся в том же самом блоке π_i разбиения Π, если и только если $S_k \Delta S_l$ и $S_l \Delta S_k$, где Δ – отношение зависимости по данным. На разбиении $\Pi = \{\pi_1, \pi_2, , \pi_v\}$ определим отношения частичного порядка $\alpha.$, $\alpha\wedge$:, и α^o следующим образом. Для $i \ne j$:

1) $\pi_i \alpha \pi_j$ тогда и только тогда, если существуют $S_k \in \pi_i$ и $S_l \in \pi_j$ такие, что $S_k \delta S_l$, где δ – отношение истинной зависимости;

2) $\pi_i \alpha\wedge \pi_j$ тогда и только тогда, если существуют $S_k \in \pi_i$ и $S_l \in \pi_j$ такие, что $S_k \bar{\delta} S_l$, где $\bar{\delta}$ – отношение антизависимости;

3) $\pi_i \alpha^o \pi_j$ тогда и только тогда, когда существуют $S_k \in \pi_i$ и $S_l \in \pi_j$ такие, что $S_k \delta^0 S_l$, где δ^0 – отношение зависимости по выходу.

Операторы будут формировать блок π_i в разбиении, Π, если и только если между ними есть направленная зависимость, циклически повторяющаяся. Между двумя блоками имеет место частично упорядоченное отношение (а, значит, есть зависимость, и блоки не могут выполняться параллельно), если и только если найдутся хотя бы два оператора, по одному из каждого блока, между которыми существует зависимость по данным.

WPG (Weighted Partition Dependence Graph) – взвешенный граф зависимости разбиения. WPG представляет собой ориентированный ациклический граф со взвешенными вершинами. Вершины WPG представляют собой блоки, а ребра показывают наличие отношений зависимости между блоками, которые они соединяют. Каждой вершине ставится в соответствие некоторое значение – вес вершины (блока). Под весом блока (операции) понимается время выполнения этого блока (операции) на соответствующем процессоре. Важной характеристикой WPG является то, что каждый WPG содержит только одну входную вершину (стартовый блок) и только одну выходную вершину (конечный блок). Входная вершина – это вершина, которая не имеет ни одного входящего ребра, а выходная – ни одного исходящего ребра. Если при построении WPG определяется более одного стартового (или конечного) блока, то добавляется формальный (пустой) блок вместе с соответствующими ребрами так, чтобы сделать его стартовым (или конечным).

```
DO I = 1, N
  DO J = 1, M
     Оператор1
     Оператор2
        ...
     ОператорK
   END DO
END DO
```

Рис. 3. Идеально вложенный цикл

Реализация операторного разбиения – задача достаточно сложная, в связи с чем для разбиения в [1–3, 10–11] были выбраны только идеально вложенные циклы (Perfectly Nested Loop). Под идеальными вложенными циклами понимаются циклы вида, показанного на рис. 3. Поэтому в модуле 1.1 (рис. 1 и 2) производится поиск

идеально вложенных циклов в исходной программе. Для каждого из найденных вложенных циклов выполняются модули 1.2–1.4. В модуле 1.2 проводится анализ зависимостей по данным между операторами тела внутри одного цикла и дается описание графа G зависимостей между операторами в цикле в виде двух множеств: V – множество вершин (операторов) и E – множество ребер (связей между операторами). В [9] истинная зависимость по данным между двумя последовательными операторами S_k и S_h (S_k расположен перед S_h) обозначается ребром, направленным от S_k к S_h, а антизависимость – ребром, направленным от S_h к S_k, поэтому граф G может содержать сильно связанные компоненты, поиском которых занимается модуль 1.3. Операторы внутри идеально вложенного цикла объединяются в блоки (каждая сильно связанная компонента – один блок). Модуль 1.4 описывает предварительный WPG в виде двух множеств: множество вершин (блоков операторов), множество ребер (связей между блоками). Каждый блок имеет ряд характеристик, таких как вес, порядок выполнения и др. (описание модуля 2, табл. 2), которые будут найдены в дальнейшем (модули 3 и 4), поэтому WPG и назван предварительным.

В работах [1–3, 10, 11] для реализации модулей 1.1 и 1.2 используется Polaris – система, предназначенная для выявления параллелизма в циклах и преобразовывающая исходную программу во внутреннее представление. Внутреннее представление состоит из управляющего графа программы, расширенной информации об операторах присваивания и графов зависимости по данным для каждого гнезда циклов, а также в него входит информация обо всех переменных, метках и массивах, использующихся в программе [14]. В [1–3, 10, 11] блоки, возникшие в результате зависимости управления, разбивают заново, конвертируя зависимость управления в зависимость по данным. После чего идет разбиение цикла на блоки и строится предварительный WPG.

Модуль 2. Построение графа гиперблоков

Введем некоторые понятия, необходимые при рассмотрении модуля 2.

HBG (Hyper Block Graph) – граф гиперблоков, представляет собой направленный ациклический граф, вершинами которого являются гиперблоки, а множество ребер описывается порядком расположения гиперблоков в программе. Каждой вершине такого графа (кроме начальной и конечной) инцидентно по два ребра (одно входящее и одно исходящее). Начальной и конечной вершине инцидентно по одному ребру (начальной – исходящее, а конечной – входящее). Основные элементы HBG представлены в табл. 2.

В модуле 2.1 конструируются наименьшие элементы HBG – блоки. Сначала запускается модуль 2.1.1, где последовательно анализируются операторы исходной программы (те, которые не обработаны в модуле 1). При этом возможны следующие варианты: 1) рассматриваемый оператор является оператором DO; 2) рассматриваемый оператор является оператором IF; 3) рассматриваемый оператор не является ни оператором DO, ни оператором IF.

Таблица 2. Основные элементы HBG и их характеристики

Наименование элемента	Содержание	Атрибуты, характеризующие элемент		
		Атрибут	Значение и описание атрибута	
Гиперблок HB (Hyper Block)	Набор блоков	тип	1	«Гиперблок с WPG» может содержать блоки циклические, нециклические и формальные
			2	«без WPG» – содержит блоки только граничного типа
		тело	перечень всех блоков, которые содержатся в гиперблоке	
		WPG	только для блоков соответствующего типа	

Продолж. табл. 2

<table>
<tr><td rowspan="9">Блок</td><td rowspan="9">Набор операторов</td><td>идентификатор</td><td colspan="2">Индивидуальный номер блока</td></tr>
<tr><td rowspan="4">тип</td><td>1</td><td>циклический – содержит цикл, за пределами цикла в блоке операторов нет</td></tr>
<tr><td>2</td><td>циклический – блок не содержит циклов</td></tr>
<tr><td>3</td><td>формальный – блок не содержит операторов вообще</td></tr>
<tr><td>4</td><td>граничный – блок может содержать переход (GOTO) к оператору другого блока, строку с несбалансированной открывающей операторной скобкой (If Условное выражение Then или ELSE или DO Счетчик=Выражение1,Выражение2), несбалансированную закрывающую операторную скобку (End If или End DO)</td></tr>
<tr><td>тело</td><td colspan="2">набор операторов, входящих в блок</td></tr>
<tr><td rowspan="4">Сведения для планирования</td><td colspan="2">вес блока (оценивается в модуле3)</td></tr>
<tr><td colspan="2">порядок выполнения блока (определяется в модуле 4)</td></tr>
<tr><td colspan="2">отношения зависимости</td></tr>
<tr><td colspan="2">и другие</td></tr>
</table>

В первом случае все операторы, начиная с данного оператора DO и до соответствующего ему End DO включительно, включают в один блок и определяют тип данного блока как циклический. Дальнейший перебор операторов продолжается со следующего после End DO оператора.

Во втором случае все операторы, начиная с данного оператора IF и до соответствующего ему End IF включительно, включают в один блок и определяют тип этого блока как нециклический. Дальнейший перебор операторов продолжается со следующего после End IF оператора.

В третьем случае все операторы добавляются в один и тот же блок до тех пор, пока не встретится новый оператор DO или IF. Тип такого блока – нециклический.

Поскольку внутри каждого блока, который содержит цикл DO или условный оператор IF, могут быть найдены еще циклы DO, то для их поиска вызывается модуль 2.1.2. Если они найдены, то блок разбивается повторно: для открывающей и закрывающей операторных скобок создают отдельные блоки, а для тех операторов, которые находятся между операторными скобками, рекурсивно запускают модуль 2.1.1. Этот процесс продолжается, пока не будут рассмотрены все нерассмотренные в модуле 1 операторы исходной программы.

В модуле 2.2 среди всех блоков выявляются блоки, которым присваивается граничный тип. Блок граничного типа должен удовлетворять хотя бы одному из следующих условий*:*

– содержать оператор STOP;

– содержать безусловный переход GOTO к операторам других блоков;

– содержать целевые операторы, т.е. те, на которые указывает GOTO;

– содержать только открывающие или закрывающие операторные скобки.

Модуль 2.3 предназначен для объединения блоков в гиперблоки, при этом каждому гиперблоку присваивается соответствующий тип: гиперблок с WPG или гиперблок без WPG. Просматриваются все блоки по порядку. Первый блок вносится в первый гиперблок. При этом, если и первый и следующий блок будут оба граничного или оба не граничного типа, то следующий блок добавляется в тот же гиперблок, что и первый блок, иначе сле-

дующий блок добавляется в новый гиперблок, а предыдущему гиперблоку назначается тип без WPG (если в него входят блоки граничного типа), иначе с WPG. Процесс продолжается, пока не будут рассмотрены все блоки.

Модуль 2.4 описывает WPG для всех гиперблоков, которые имеют тип ”с WPG“. Просматриваются все гиперблоки из HBG. Если гиперблок имеет тип ”с WPG“, то просматриваются все блоки, в него входящие, и заполняются два множества: множество ребер и множество вершин WPG. Для заполнения множества ребер необходимо проанализировать зависимости по данным между блоками. Для дальнейшей обработки важно, чтобы WPG имел только один вход (стартовый блок) и один выход (конечный блок). Поэтому в модуле 2.4 для WPG, имеющего более, чем один стартовый блок, добавляется формальный (пустой) блок, который делают родительским по отношению ко всем стартовым блокам, после чего этот формальный блок становится стартовым. Аналогично поступают и в случае, когда WPG содержит более одного конечного блока. В этом случае добавленный пустой блок делают дочерним по отношению ко всем завершающим, после чего он становится единственным конечным блоком.

Модуль 3. Оценка веса

В модуле 3 для гиперблоков, которые имеют тип «с WPG», оценивается вес каждого блока. Для оценки веса блока необходимо оценить вес каждой операции, в него входящей.

Напомним, что под весом блока (операции) понимается время выполнения этого блока (операции) на соответствующем процессоре. Вес представляется в виде пары чисел *W= (PHW, PMW),* где *PHW* – вес при выполнении блока отдельно на P.Host, *PMW* – на P.Mem. Под весом понимается время выполнения данного блока на данном процессоре. При оценке веса интегрируются два подхода: сначала анализ кода и поиск по таблице веса операций (модуль 3.1), а если блок содержит неизвестные операции, то включается механизм для оценки веса неизвестных операций (модуль 3.3). Полученные при этом веса добавляются в таблицу (модуль 3.4). После чего оценивается вес всего блока (модуль 3.5) [3, 10].

Модуль 4. Возвратный механизм планирования

Определим некоторые важные понятия, используемые в дальнейшем.

Метод критического пути (МКП). Пусть в задаче требуется найти минимальное время, необходимое для реализации некоторого проекта, состоящего из большого числа этапов. Каждый этап можно изобразить вершиной графа, при этом если этап b_i должен непосредственно предшествовать этапу b_j, то от вершины b_i проводится дуга к вершине b_j. Получим ориентированный ациклический граф. При этом вершинам графа может приписываться вес (например, время выполнения этапа). Дуги графа также могут быть взвешены (вес дуги – минимальное время задержки между выполнением соответствующих этапов). Для нахождения минимального времени выполнения проекта нужно в графе найти самый длинный путь (здесь имеется в виду путь с самым большим весом, стоимостью) между вершиной b_s, изображающей начало, и вершиной b_e, изображающей завершение всех необходимых для реализации проекта работ. Самый длинный путь называется критическим путем, так как этапы, относящиеся к этому пути, определяют полное время реализации проекта, и всякая задержка с началом выполнения любого из этих этапов приведёт к задержке выполнения проекта в целом [15].

Нахождение длины критического пути графа часто является одной из важных задач при распараллеливании алгоритма, поскольку критический путь является минимальной высотой всех возможных ярусно-параллельных форм (ЯПФ) графа.

Ярусно-параллельная форма (ЯПФ) графа – это деление вершин ориентированного ациклического графа на перенумерованные подмножества V_i такие, что если дуга e направлена от вершины $v_1 \in V_j$ к вершине $v_2 \in V_k$, то обязательно $j < k$. При этом множест-

ва V_i называются ярусами ЯПФ, i – номером соответствующего яруса, количество вершин в ярусе — его шириной.

Высота ЯПФ – это количество ярусов в ЯПФ. *Ширина ЯПФ* – это максимальная ширина её ярусов.

В ярусы объединяются операторы, для выполнения которых необходимы значения, вычисляемые на предыдущих ярусах.

Для нахождения критического пути в графе пользуются следующим утверждением: вершина b_i находится на критическом пути тогда и только тогда, когда выполняется равенство

$$rank_u(b_i) + rank_d(b_i) = rank_u(b_s), \quad (1)$$

где b_s – стартовая вершина WPG, а b_i при этом называют *вершиной критического пути* [10];

$rank_u(b_i)$ и $rank_d(b_i)$ – соответственно восходящий (upward rank) и нисходящий ранги (downward rank) вершины b_i [16]. Эти значения определяются рекурсивно:

$$\begin{aligned} rank_u(b_i) &= W(b_i) + \max_{b_j \in \ succ(b_i)} (rank_u(b_j) + c_{ij}), \\ rank_d(b_i) &= \max_{b_j \in pred(b_i)} \{rank_d(b_j) + W(b_j) + c_{ji}\}, \end{aligned} \quad (2)$$

где $succ(b_i)$ и $pred(b_i)$ представляют соответственно все последующие (дочерние) и все предшествующие (родительские) вершины по отношению к b_i, $W(b_i)$ – вес вершины b_i, c_{ij} – вес ребра, направленного из i-той вершины к j-той. При этом вначале просматривают граф в направлении от начальной до конечной вершины (сверху вниз), определяя нисходящий ранг $rank_d$ каждой вершины, начальное значение $rank_d$ *для* стартовой вершины равно 0. Далее граф просматривается снизу вверх (от конечной до стартовой вершины) и определяется восходящий ранг $rank_u$ каждой вершины, начальным значением здесь служит $rank_u$ завершающей вершины, который равен весу этой вершины.

Из (2) видно, что $rank_d(b_i)$ – это наибольший путь от стартовой вершины до вершины b_i, не включая вес (время выполнения) данной вершины непосредственно, а $rank_u(b_i)$ – длина критического пути от b_i до конечной вершины, включая вес данной вершины. Таким образом, в (1) $rank_u(b_s)$ – это длина критического пути.

В модуле 4 в цикле анализируются на предмет параллельного выполнения все гиперблоки с WPG и WPG идеально вложенных циклов. Для планирования блоков в каждом WPG пользуются МКП.

В модуле 4.1 проводится инициализация необходимых переменных, обнуляется значение порядка выполнения (O) и обратного порядка выполнения (RO) для каждого блока из WPG.

В модуле 4.2 для каждого блока вычисляются порядок выполнения и его нисходящий ранг. Порядок выполнения находится по индуктивному принципу: 1) стартовый блок имеет порядок выполнения 1; 2) если у родительского блока порядок выполнения равен i, то у дочернего – $i+1$.

В модуле 4.4 находится восходящий ранг для каждого блока. Для поиска восходящего и нисходящего рангов пользуются формулами (2), причем веса ребер считаются равными 0 ($c_{ij} = 0$). В [10, 11] в качестве веса блока b_i – $W(b_i)$ используется его вес для процессора памяти $PMW(b_i)$ и формулы (2) принимают вид

$$rank_u(b_i) = PMW(b_i) + \max_{b_j \in succ(b_i)}(rank_u(b_j)),$$
$$rank_d(b_i) = \max_{b_j \in pred(b_i)}\{rank_d(b_j) + PMW(b_j)\}. \quad (3)$$

В [5] в качестве $W(b_i)$ предлагается использовать наименьший из двух весов $PHW(b_i)$, $PMW(b_i)$, т. е. $W(b_i) = \min\{PHW(b_i); PMW(b_i)\}$.

В модуле 4.4 на основе условия (1) определяются блоки критического пути и ведется подсчет количества разделов (секций) WPG (по количеству критических блоков).

В модуле 4.5 планируют критический блок внутри каждого раздела WPG на наиболее подходящий процессор. Для этого сравнивают вес данного блока для хост-процессора и для процессора памяти. В один раздел WPG входят все блоки, которые имеют порядок выполнения больший или равный порядку выполнения критического блока из данного раздела, но меньший чем порядок выполнения критического блока из следующего раздела. Фактически разбиение WPG на разделы – это представление графа в ярусно-параллельной форме.

В модуле 4.6 блоки, принадлежащие одному и тому же разделу, разделяют на несколько внутренних волновых фронтов (в один внутренний волновой фронт попадают блоки, имеющие одинаковый порядок выполнения) и внутри каждого волнового фронта планируют блоки на подходящий процессор.

В модуле 4.7 генерируют структурную схему выполнения блоков WPG в виде

$$CPS = \{CPS_1, CPS_2, ..., CPS_i\},$$

где CPS *(Critical Block Scheduling)* – это структурная схема выполнения исходной программы;

CPS_i – структурная схема выполнения i-того раздела (яруса) WPG.

$$CPS_i = \{CP_i, IWF_i\},\ CP_i = \{\Pr ocessor(b_{critical})\},$$

где $\Pr ocessor$ – это PH *(хост-процессор)* или $PM1$*(процессор памяти 1)* показывает, какой из процессоров назначен для выполнения критического блока $b_{critical}$ из i-того раздела WPG.

IWF_i – список внутренних фронтов волн для i-того раздела WPG.

$$IWF_i = \{iwf_{i1}, iwf_{i2}, ..., iwf_{ij}\},$$

где iwf_{ij} – внутренний фронт волн;

$iwf_{ij} = \{PH(b_a), PH(b_b), PH(b_c), ...\}$ – эта запись означает, что во внутреннем фронте волны ij блок b_a будет назначен хост-процессору, блок b_b – процессору памяти 1, блок b_c – процессору памяти 2 и так далее.

В модуле 4.8 проверяется, не превышает ли число процессоров, используемых в структурной схеме (из модуля 4.7), общего числа процессоров в системе. Если превышает, то структурная схема выполнения корректируется.

4. Выводы

Результаты анализа структуры алгоритма распределения приложения для PIM-системы позволяют сделать следующие выводы.

1. Укрупненный алгоритм, приведенный на рис. 1, является достаточно сложным и трудоемким, требующим больших затрат времени на его выполнение. Цепочка вычислений слишком длинная, алгоритм содержит 22 модуля, почти каждый из них представляет собой программу, содержащую огромное количество переборов, и подпрограммы, которые в свою очередь тоже включают огромное количество операций. В силу сложности реали-

зации такого алгоритма проводится анализ и выявление зависимостей по данным только для некоторых конструкций исходной программы (для идеально вложенных циклов), другие конструкции или пропускаются вообще, как в [1–3, 10–11], или при разбиении на блоки этих конструкций не проводятся анализ зависимостей между операторами исходной программы (модуль 2).

2. Выявление зависимостей по данным как в циклах, так и при ссылках на массивы, тоже достаточно сложная задача. Причем часто массивы и циклы в программе встречаются вместе и в индексах массивов используют параметры циклов, что делает выявление всех зависимостей по данным между операторами практически неразрешимой задачей.

3. В модуле 1.3 осуществляется поиск сильно связанных компонент графа, процедура поиска нуждается в доработке (алгоритм приведен в [9]), поскольку, сколько раз ее запускать и с какими параметрами, решает пользователь. Также не для всех графов этот алгоритм приводит к правильному результату.

Для поиска сильно связанных компонент в теории графов существует ряд алгоритмов: алгоритмы Косарайю, Габова и Тарьяна [17], каждый из этих алгоритмов находит все сильно связанные компоненты графа за линейное время (время исполнения алгоритма пропорционально числу ребер графа), каким-либо из них можно воспользоваться для поиска сразу всех сильно связанных компонент графа.

4. В модулях 4.2 и 4.3 предполагается, что в описании WPG блоки расположены в произвольном порядке, но, возможно, если изначально найти порядок выполнения каждого блока и пронумеровать блоки так, чтобы выполнялось условие: если блок имеет некоторый номер i, то все его родительские блоки имеют номера меньше, чем i, а дочерние блоки – больше, чем i, то можно уменьшить количество переборов в модулях 4.2 и 4.3. При этом в модуле 4.3 может не понадобиться вычислять обратный порядок выполнения, а также наименьший и наибольший обратный порядки выполнения всех дочерних блоков для каждого блока, если просматривать блоки в порядке уменьшения их номера.

5. В модуле 4.4 (рис. 1 и 2) не учтен тот факт, что в графе может оказаться несколько критических путей. Поскольку модуль 4.4 возвращает множество всех критических блоков и WPG разбивается на разделы по количеству критических блоков, то в результате будет найдено множество критических блоков, принадлежащих разным критическим путям, и WPG будет неправильно разбит на разделы.

6. Можно существенно упростить алгоритм распределения приложений пользователя для PIM-системы, используя особенности, отличающие ее от компьютерных систем с классической архитектурой. К таким особенностям относятся следующие:

– наличие большого количества средств обработки непосредственно на кристалле с присоединенными к ним устройствами (банками) памяти, что позволяет одновременно выполнять большое количество операций и даже участков программ;

– использование на том же кристалле так называемого ведущего процессора, который берет на себя функции реализации наиболее трудоемких операций алгоритма пользовательской задачи и управляет работой всех других процессорных элементов с прикрепленной к ним памятью;

– PIM по своему принципу построения реализует иерархическую структуру ресурсов как средств обработки (например, ведущий процессор наверху, а внизу процессоры памяти, упрощенные процессорные ядра), так и памяти (основная память ВП, его кэш-память, основная память каждого ПЯ, их кэш-память и дополнительная внешняя память), что позволяет предварительно распределить участки программы пользователя как в зависимости от возможностей средств обработки, так и в зависимости от использования на разных уровнях памяти данных;

– большое количество параллельно использующихся средств обработки (ПЯ) позволяет не заботиться о достижении точного баланса загрузки как всех процессоров, так и

памяти, поскольку при глубоком распараллеливании один или несколько несбалансированных ресурсов не сыграет важной роли в достижении высокой производительности.

Используя перечисленные особенности, можно сформулировать основные принципы подхода к распределению приложений для PIM-системы:

1. Принцип соответствия реализации алгоритма системе команд процессора, на котором он должен быть выполнен.

2. Принцип максимального использования иерархической структуры ресурсов PIM-системы.

3. Принцип равноправия используемых ресурсов нижнего уровня, обеспечивающий возможность замены множества процессорных элементов и модулей памяти на один эквивалентный комплексный ресурс, содержащий один процессор с системой команд объединенных ПЯ и скоростью работы, увеличенной в k (количество ПЯ) раз, с емкостью памяти, равной сумме емкостей памяти всех ПЯ.

4. Принцип укрупненной балансировки загрузки процессоров по специфическим критериям.

5. Принцип реконфигурируемости, обеспечивающий настройку архитектуры и структуры PIM-системы на оптимальное решение пользовательской задачи.

Основные положения подхода для реализации указанных принципов при распределении приложений для PIM-системы изложены в [18–20].

СПИСОК ЛИТЕРАТУРЫ

1. Tsung-Chuan Huang A Parallelizing Framework for Intelligent Memory Architectures [Электронный ресурс] / Tsung-Chuan Huang, Slo-Li Chu [Електронний ресурс] // Proc. of The Seventh Workshop on Compiler Techniques for High-Performance Computing. – Taiwan, 2001. – March 15–16. – P. 96 – 102. – Режим доступу: http://parallel.iis.sinica.edu.tw/cthpc/7th/ 03CTHPC2001-Hardcopy.pfd.

2. Slo-Li Chu Exploiting Application Parallelism for Processor-in-Memory Architecture / Slo-Li Chu, Tsung-Chuan Huang [Електронний ресурс] // Proc. of National Computer Symposium. – Taiwan, 2003. – December 18–19. – P. 293 – 303. – Режим доступу: http://dspace.lib.fcu.edu.tw/bitstream/2377/564/ 1/OT_1022003305.pdf.

3. Slo-Li Chu SAGE: An Automatic Analyzing System for a New High-Performance SoC Architecture – Processor-in-Memory / Slo-Li Chu, Tsung-Chuan Huang // Journal of Systems Architecture: the EUROMICRO Journal. – 2004. – Vol. 1. – P. 1 – 15.

4. Slo-Li Chu POERS: A Performance-Oriented Energy Reduction Scheduling Technique for a High-Performance MPSoC Architecture // 11th International Conference on Parallel and Distributed Systems (ICPADS 2005), (20–22 July 2005). – 2005. – Vol. 2. – P. 699 – 703.

5. Hwa-Jyh Jean Designing New Scheduling Mechanisms for Processor-in-Memory Systems [Електронний ресурс] / Hwa-Jyh Jean // Department of Electrical Engineering National Sun Yat-Sen University. – 2001. – Режим доступу: http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/view_etd?URN=etd-0613101-192127.

6. Kun-En Hsieh The Implementation of Code Generator for Processor-in-Memory Systems [Електронний ресурс] / Kun-En Hsieh // Department of Electrical Engineering National Sun Yat-Sen University, 2002. – Режим доступу: http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/view_etd?URN=etd-0808102-153641.

7. Ming-Yong Chen The Implementation of Task Evaluation and Scheduling Mechanisms for Processor-in-Memory Systems [Електронний ресурс] / Ming-Yong Chen // Department of Electrical Engineering National Sun Yat-Sen University. – 2002. – Режим доступу: http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/view_etd?URN=etd-0809102-182526.

8. Jyh-Chiang Huang The Design of an Effective Load-Balance Mechanism for Processor-in-Memory Systems [Електронний ресурс] / Jyh-Chiang Huang // Department of Electrical Engineering National Sun Yat-Sen University. – 2002. – Режим доступу: http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/view_etd?URN=etd-0826102-154856.

9. Ying-Bo Liu The Design of a New Program Decomposition Mechanism for Processor-in-Memory Systems [Електронний ресурс] / Ying-Bo Liu // Department of Electrical Engineering National Sun Yat-Sen University. – 2002. – Режим доступу:
http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/view_etd?URN=etd-0826102-153046.
10. Slo-Li Chu Critical Block Scheduling: a Thread-Level Parallelizing Mechanism for a Heterogeneous Chip Multiprocessor Architecture [Електронний ресурс] / Slo-Li Chu // Languages and Compilers for Parallel Computing: 20th International Workshop (USA, October 11–13, 2007). – 2007. – P. 261 – 275. – Режим доступу: http://polaris.cs.uiuc.edu/lcpc07/accepted/51_Final_Paper.pdf.
11. Slo-Li Chu Toward to Utilize the Heterogeneous Multiple Processors of the Chip Multiprocessor Architecture / Slo-Li Chu // Proc. International Conference, EUC 2007. – 2007. – December. – P. 234 – 246.
12. Wikipedia [Електронний ресурс]. – Режим доступу:
http://en.wikipedia.org/wiki/Data_dependency#Data_dependencies.
13. Баканов В.М. Параллельные вычисления: учебное пособие / Баканов В.М. – Москва: МГУПИ, 2006. – 124 с.
14. The Polaris Internal Representation / Keith A. Faigin, Jay P. Hoeinger, David A. Padua [et al.] // International Journal of Parallel Programming. – 1994. – Vol. 22(5). – P. 553 – 586.
15. Кристофидес Н. Теория графов. Алгоритмический подход / Кристофидес Н. – М.: Мир, 1978. – 432 с.
16. Olivier B. The iso-level scheduling heuristic for heterogeneous processors [Електронний ресурс] / B. Olivier, B. Vincent, R. Yves // Proc. of 10th Euromicro workshop on parallel, distributed and network-based processing, 2002. – P. 335 – 350. – Режим доступу:
http://lara.inist.fr/bitstream/2332/760/1/LIP-RR2001-22.pdf.
17. Седжвик Р. Фундаментальные алгоритмы на C++. Часть 5: Алгоритмы на графах / Седжвик Р.; пер. с англ. – СПб.: ДиаСофтЮП, 2002. – 496 с.
18. Елисеева Е.В. Метод распределения приложений для оптимизации вычислений в компьютерной системе типа „процессор-в-памяти” / Елисеева Е.В. // Праці міжнародного симпозіуму „Питання оптимізації обчислень (ПОО-XXXV)”. – К.: Інститут кібернетики імені В.М. Глушкова НАН України, 2009. – Т. 1.– С. 227 – 231.
19. Яковлев Ю.С. Основные принципы и методика распределения приложений в сложных компьютерных системах типа “процессор-в-памяти” / Ю.С. Яковлев, Е.В. Елисеева // УСиМ. – 2009. – № 6. – С. 56 – 63.
20. Елисеева Е.В. Реализация служебной функции средств поддержки вычислительного процесса интеллектуальной памяти компьютерных систем / Е.В. Елисеева // Інформаційні технології та комп’ютерна інженерія. – 2009. – № 3. – С. 43 – 47.

Стаття надійшла до редакції 22.06.2010