УДК 004.3; 004.4; 681.5

А.В. Бакулев

АЛГОРИТМ СИНТЕЗА ПАРАЛЛЕЛЬНОЙ РЕАЛИЗАЦИИ ПОСЛЕДОВАТЕЛЬНОЙ ПРОГРАММЫ ДЛЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ, ПОСТРОЕННЫХ НА БАЗЕ МНОГОЯДЕРНЫХ ПРОЦЕССОРОВ

Предложен алгоритм синтеза параллельной реализации последовательной программы, ориентированный на исполнение в среде вычислительных систем, построенных на базе многоядерных процессоров. Алгоритм базируется на графовой модели последовательной программы. Приведен пример работы алгоритма. Показаны направления для дальнейших исследований.

Ключевые слова*: параллельные вычисления, многоядерные процессоры, распараллеливание программ, графовые модели программ.*

Введение. Развитие технологий программирования для персональных компьютеров с точки зрения *повышения производительности программ* традиционно происходило в основном экстенсивными методами. По большей части разработчики программного обеспечения (ПО) уповали на постоянное совершенствование аппаратных ресурсов компьютера. Таким образом, очередная замена процессора с увеличенной тактовой частотой позволяла получить ускорение работы программ без малейшего их изменения. В настоящее время достигнут фактический предел роста тактовой частоты процессора, и дальнейшее увеличение его производительности становится возможным только за счет перехода к ***многоядерной*** архитектуре [1], по существу к ***параллельной*** архитектуре (на сегодняшний день уже широко распространены процессоры с четырьмя ядрами, — в перспективе количество ядер может измеряться сотнями). Подобная технологическая революция в архитектуре персонального компьютера предоставляет существенный потенциал для роста производительности ПО, однако для реализации этого потенциала необходимы столь же серьезные революционные изменения современных технологий программирования, а также решение задачи адаптации огромного объема существующего последовательного ПО для эффективного выполнения в параллельной вычислительной среде.

Вопросы параллельного программирования, а также автоматизации распараллеливания последовательного ПО для определенного класса параллельных архитектур имеют уже долгую историю [2,3]. Проблема заключается в том, что, как правило, решались они в привязке к определенным типам архитектур параллельных компьютеров. Подобный подход требовал создания специализированного ПО для каждой разновидности программно–аппаратной платформы параллельного вычислителя. Со временем появились универсальные библиотеки параллельного программирования, такие как MPI для кластеров и систем с распределённой памятью, а также OpenMP для систем с общей памятью. Однако при написании параллельной программы для обеспечения эффективности работы полученной реализации и достижения сбалансированности нагрузки на процессоры приходится учитывать массу деталей, связанных с параметрами конкретной целевой параллельной архитектуры, что существенно ограничивает переносимость программ. Ряд отечественных и зарубежных разработок: DVM [4], PARUS [5], Т-система [6], Cilk [7] предлагают автоматизировать процесс разработки параллельных программ с учетом их дальнейшего масштабирования, однако они либо строго нацелены на использования конкретных языков высокого уровня (C, Fortran), либо предлагают вообще отказаться от императивной парадигмы в пользу функционального программирования (Т-система).

Использовать этот богатый опыт разработки параллельных программ в области универсального ПО для персональных компьютеров напрямую невозможно, учитывая многообразие разновидностей операционных систем, языков программирования, аппаратных платформ компьютеров и огромный багаж последовательных программ, представленных как в исходных текстах на различных языках высокого уровня, так и в объектной и исполняемой форме. Таким

образом, задача эффективного переноса последовательных программ и организации мобильных параллельных вычислений для вычислительных систем на основе многоядерных процессоров является ***актуальной*** и востребованной.

Современное решение переноса последовательных программ заключается в использовании технологий виртуализации вычислений [8] и использования промежуточного кода программ. Так, в современной промышленной индустрии разработки ПО доминируют технологии ***Java*** и ***.Net***, основанные на концепции виртуальной машины, однако они плохо приспособлены для решения задачи эффективного распараллеливания существующих последовательных программ.

Целью работы является разработка алгоритма синтеза параллельной реализации последовательной программы на основе предлагаемой концепции ***виртуальной параллельной машины*** и *параллельного промежуточного представления программ*. В рамках данной концепции перенос существующего ПО разбивается на два этапа. На первом этапе происходит конвертация (трансляция) последовательных программ в параллельный промежуточный код. ***Научная новизна*** данного подхода состоит в том, что предлагается оригинальное промежуточное представление программ, не ограничиваемое конкретным языком высокого уровня, с возможностью преобразования исполняемых программ для различных операционных систем и компьютерных архитектур. Кроме того, на этом этапе происходит предварительное распараллеливание программ (выявление потенциального параллелизма), не учитывающее пока конкретных условий их исполнения на целевой вычислительной системе. Этим обеспечивается хорошая переносимость промежуточного хода в среду будущих параллельных вычислений.

На втором этапе происходит исполнение параллельного промежуточного кода ядром виртуальной параллельной машины на конкретной многоядерной вычислительной системе. Для этого производится привязка параллельного кода к архитектурным особенностям целевой вычислительной системы, например, с учетом количества процессорных ядер, их загруженности, параметров производительности и т. д. Результатом является окончательный план распределения вычислений параллельного промежуточного кода на всех доступных процессорных ядрах целевого компьютера.

В данной статье предлагается алгоритм синтеза параллельной реализации последовательной программы на втором этапе её распараллеливания ядром виртуальной параллельной машины. В соответствии с предложенной концепцией исходными данными алгоритма являются максимальная параллельная форма последовательной программы, полученная на первом этапе трансляции в параллельный промежуточный код, и параметры целевой вычислительной системы (количество процессорных ядер). Это позволяет снизить трудоемкость задачи распараллеливания в сравнении с известными решениями [9, 10, 11]. Что особенно важно, так как вычислительные затраты, необходимые на преобразование последовательной программы, на первом этапе носят разовый характер, а параллельное исполнение промежуточного кода ядром виртуальной машины производится многократно.

Постановка задачи. Наиболее простой моделью последовательной программы является её представление в виде графа без циклов и вероятностных ветвлений. Изучение свойств алгоритмов и программ, описанных с помощью графовых моделей, приведено в работах Корячко В.П. [12], Касьянова В.Н. [13], в частности в связи с задачей распараллеливания программ, — Воеводина В.В., Воеводина Вл.В. [3], Корячко В.П., Скворцова С.В., Телкова И.А. [14]. В приведенных работах доказано, что именно графовые модели позволяют наиболее полно исследовать свойства последовательной программы и её параллельной реализации. Поэтому использование графовой модели при разработке алгоритма синтеза параллельной реализации последовательной программы является обоснованным.

Рассмотрим граф программы $A=(V,U,\Im)$, где макровершины графа V представляют линейные участки программы [3], а дуги — информационные и управляющие связи макровершин. Вершины взвешены временем выполнения $\tau(v_i)\in\Im \mid v_i\in V$ на одном процессорном элементе (в тактах процессора).

Для удобства дальнейшего изложения введем две дополнительные псевдовершины $v_0\in V$ — **начальную** и $v_{N+1}\in V \mid N=|V|$ — **конечную,** полагая, что $\tau(v_0)=\tau(v_{N+1})=0$. Пусть вершина v_0 не содержит ни одной входящей дуги $(v_i,v_0)\notin U \mid i=\overline{1,N}$ и соединяет все начальные вершины дугами $(v_0,v_i)\in U \mid (v_j,v_i)\notin U, i=\overline{1,N}, j=\overline{1,N}$. Напротив, из вершины v_{N+1} не выходит ни

одной дуги $(v_{N+1}, v_i) \notin U \mid i = \overline{1,N}$, но v_{N+1} связывает все конечные вершины дугами $(v_i, v_{N+1}) \in U \mid (v_i, v_j) \notin U, i = \overline{1,N}, j = \overline{1,N}$.

Для представления распределенного по процессорным элементам системы вычислительного процесса во времени введем понятие *пространственно-временной решетки*. Пусть *решетка* L^n — это совокупность точек n-мерного пространства с целыми координатами. Тогда *пространственно-временная решетка* — L^2-решётка, заданная на пространстве процессорных элементов, образованная парой ортогональных осей координат.

В связи с однородностью многоядерной вычислительной системы все процессорные ядра считаются равноправными и обладают идеентичными временными характеристиками. Это позволяет перейти к L^2-решётке, именуемой PT-решёткой и рассматриваемой далее. Ось абсцисс — P-ось образована номерами процессорных элементов системы. Ось ординат — T-ось представляет время работы вычислительной системы в тактах процессора.

Укладкой $\pi(A)$ графа программы $A = (V, U, \Im)$ в PT-решётку называется отображение $\pi: V \to L^2$ множества V вершин графа в множество L^2 узлов решётки, определяющее проекции вершин:

$$\pi(v_i) = (v_i^p, v_i^t), i = \overline{1,N},$$

где v_i^p — проекция на ось *P*, v_i^t — проекция на ось *T*.

Моменты времени начала выполнения и окончания i-го линейного участка соответственно определяются как $t_H(v_i) = v_i^t$, $t_K(v_i) = v_i^t + \tau(v_i)$. Длительность реализации всей программы представляет высоту укладки графа $H = T(\pi(A)) = v_{N+1}^t - 1$.

Для данной укладки $\pi(A)$ можно определить сечение t_S по T-оси как множество вершин $V(t_S)$, требующих ресурсов процессорного времени на *s*-м такте $V(t_S) = \{v_i \mid v_i^t = t_S, v_i \in V\}$.

Шириной t_S-сечения укладки $W_S(\pi(A)) = |V(t_S)|$ называется общее количество вершин, находящихся в данный момент времени на исполнении, что определяет необходимое количество процессорных ядер, задействованных на t_S-м такте. Величина:

$$W(\pi(A)) = \max_{S=1}^{H} \{W_S(\pi(A))\} \quad (1)$$

задаёт минимально необходимое число процессорных ядер, требуемых для реализации укладки графа $\pi(A)$, и называется ***шириной укладки*** $\pi(A)$.

Подмножество вершин, имеющих одинаковые P-проекции, представляет p_k-сечение укладки $\pi(A)$:

$$V(p_k) = \{v_i \mid v_i^p = p_k, v_i \in V\}, \quad (2)$$

т.е. объединяет вершины, исполняемые на одном процессорном ядре $p_k \in \Re$, где $\Re$ — множество процессорных элементов многоядерной вычислительной системы.

Время последовательного исполнения всех линейных участков программы на одном процессорном ядре можно рассчитать как

$$H_{\max} = \sum_{i=1}^{N} \tau(v_i).$$

Для определения временных характеристик её параллельного выполнения введем ряд величин. Для каждой вершины графа рассчитаем предварительно $E(v_i)$ ***наиболее ранний*** срок выполнения как длину критического пути от начальной вершины графа v_0 к данной v_i. Величина:

$$H_{\min} = E(v_{N+1}) - 1 \quad (3)$$

определяет минимально возможное время выполнения программы в случае отсутствия ограничения на число процессоров системы (максимальный параллелизм). Тогда высота оптимальной укладки графа $\pi^*(A)$ может находиться в пределах $H_{\min} \le H^* \le H_{\max}$.

Наиболее поздний срок выполнения $L(v_i, H)$ рассчитывается для заданной высоты H укладки программы на основе длины критического пути $\lambda(v_i)$ от данной вершины v_i к конечной v_{N+1}. $L(v_i, H) = H + 1 - \lambda(v_i)$.

В случае максимального параллелизма составляет: $L(v_i, H_{\min}) = E(v_{N+1}) - \lambda(v_i)$.

Так как увеличение высоты укладки графа на t-единиц приводит к возрастанию наиболее поздних сроков для всех вершин также на t-единиц времени, получить необходимые значения наиболее поздних сроков реализации вершин при изменении высоты укладки H

можно на основе первоначально рассчитанных как $L(v_i, H) = L(v_i, H_{\min}) + (H - H_{\min})$.

Зная величины $E(v_i), L(v_i, H)$, для каждой вершины укладки можно определить следующие временные характеристики программы.

- ***Полный резерв времени***:

$$R'(v_i) = \min_j \{L(v_j, H) \mid \exists(v_i, v_j) \in U\} - E(v_i) - \tau(v_i) \geq 0 \cdot$$

Полный резерв времени определяет максимально возможные задержки выполнения вершины, которые ещё не приводят к увеличению высоты всей укладки в целом.

- ***Свободный резерв времени***:

$$R''(v_i) = \min_j \{E(v_j) \mid \exists(v_i, v_j) \in U\} - E(v_i) - \tau(v_i) \geq 0 \cdot$$

Свободный резерв времени определяет максимально возможные задержки выполнения вершины, которые не влияют на выполнение последующих, связанных с ней вершин.

- ***Независимый резерв времени***:

$$R'''(v_i) = \min_j \{E(v_j) \mid \exists(v_i, v_j) \in U\} - L(v_i, H) - \tau(v_i) \cdot$$

Независимый резерв времени определяет максимально возможные задержки вершины, которые не накладывают ограничения на время выполнения любой другой вершины графа. Отрицательное значение независимого резерва показывает, что любая задержка выполнения данной вершины заведомо накладывает ограничения на выполнение прочих.

Вершина называется ***критической***, если любая задержка в ней приводит к задержке завершения всей программы (изменению высоты укладки графа):

$$\forall v_k \mid R'(v_k) = 0 \Leftrightarrow \forall v_k \mid E(v_k) = L(v_k, H).$$

Задача составления плана реализации программы оптимального по времени исполнения в системе из $p = |\Re|$ процессорных ядер представляет ***задачу минимизации целевой функции***:

$$T(\pi(A)) \to \min,$$

на системе ограничений:

$$\begin{cases} v_i^t < v_j^t - \tau(v_i), \exists(v_i, v_j) \in U, i = \overline{1, N}, j = \overline{1, N} & (4) \\ v_i^t > v_k^t + \tau(v_k), \exists(v_k, v_i) \in U, i = \overline{1, N}, k = \overline{1, N} & (5) \\ v_i^t \geq E(v_i), i = \overline{1, N} & (6) \\ v_i^p \neq v_l^p, \exists t_s \mid v_l \in V(t_s) \wedge v_i \in V(t_s), i = \overline{1, N}, l = \overline{1, N} & (7) \\ W(\pi(A)) \leq p. & (8) \end{cases}$$

Наличие связей по управлению и данным для вершин графа накладывает ограничения на множество всех допустимых укладок $\Pi(A)$. При этом неравенство (4) задает ограничение на завершение выполнения предыдущей из двух связных вершин до начала исполнения последующей. Неравенство (5) определяет обратное условие, диктуемое невозможностью инициализации очередной вершины до момента окончания выполнения любой предыдущей, связанной с ней. Условие (6) вытекает из определения наиболее ранних сроков выполнения вершины, вычисляемых по выражению (3). Неравенство (7) накладывает пространственное ограничение на размещение вершин графа — условие несовпадения P-проекций любой пары вершин, принадлежащих одновременно какому-либо t_s-сечению укладки графа, т.е. лежащих в пересекающихся временных диапазонах. Неравенство (8) накладывает ограничение на ширину укладки (1), связанное с фиксированным числом процессорных ядер вычислительной системы.

Таким образом, процесс решения задачи сводится к нахождению среди множества допустимых укладок $\Pi(A) = \{\pi^{(k)}(A)\}$ графа $A = (V, U, \Im)$ укладок $\{\pi^*(A)\}$, имеющих минимальную высоту H^*.

Алгоритм синтеза параллельной реализации последовательной программы. Для обеспечения параллельного исполнения отдельных частей программы, на многоядерной вычислительной системе существует, по крайней мере, два способа:

1) организация разбиения программы на ряд взаимодействующих подзадач;

2) представление программы в виде одной задачи с последующим её разбиением на отдельные потоки (threads - нити).

Первый способ больше подходит для слабо связанных и относительно независимых частей выполняемой программы. Обмен данными между отдельными подзадачами реализуется специальными механизмами операционной системы, в общем случае организующими специальные каналы обмена данными между ними. При этом время, затраченное на организацию сеансов обмена, становится значимым для качественного распараллеливания программы. Следовательно, одним из критериев оптимальности распараллеливания будет являться критерий минимальной связности подзадач.

Во втором случае все потоки задачи функционируют в едином адресном пространстве, т.е. разделяют общую память. Поэтому характер и количество связей не будут определять непосредственных временных затрат по обмену. При совместном использовании общих ресурсов требуется обеспечить лишь синхрони-

зацию взаимодействующих непосредственно пар потоков. Этот способ параллелизма лучше подходит именно для многоядерных вычислительных систем, поэтому и был взят за основу.

Графовую модель представления последовательной программы, рассмотренную выше, можно использовать при организации статического распараллеливания последовательной программы на ряд потоков с учетом ограничения на общее количество процессорных ядер *p* вычислительной системы.

Решение данной задачи точными методами математического программирования является весьма трудоемким. Более предпочтительным оказывается эвристический подход, основанный на фиксации начальной укладки, в качестве которой выбирается укладка максимальной степени параллельности. Последующие преобразования ориентированы на приведение ее ширины к заданной.

Предлагаемый алгоритм содержит следующие шаги.

п.1. Построение минимальной по высоте укладки без учета ограничений на ширину $\tilde{\pi}(A)$ производится на основании предварительно вычисленных значений $E(v_i),\ v_i^t = E(v_i)$.

п.2. Если $W(\tilde{\pi}(A)) \le p$, то переход к **п.6.**

п.3. Просмотр укладки по *s*-сечениям в порядке возрастания номеров, начиная с первого. *s=1*.

п.4. Если $W_s(\tilde{\pi}(A)) - p = d > 0$, необходимо выбрать *d*-вершин для перемещения ниже $(v_i^t = v_i^t + 1)$, руководствуясь следующими правилами в порядке убывания предпочтения:

– $R'''(v_i) > 0$;

– $R''(v_i) > 0$;

– $R'(v_i) > 0$;

– вершина с меньшим числом транзитивных связей;

– вершина с меньшим весом.

Если смещение вершины влечет: $R'(v_i) < 0$, высота укладки увеличивается (*H=H+1*) и все транзитивные вершины перемещаются ниже

п.5. Если не все сечения просмотрены, то *s=s+1* и переход к **п.4**.

п.6. Конец алгоритма.

По результатам выполнения алгоритма можно перейти к параллельной реализации программы следующим образом. Графу программы $A = (V, U, \mathfrak{I})$ приведем в соответствие его реализацию – процесс: $A = (V, U, \mathfrak{I}) \to \eta_i(t)$. Оптимальная укладка графа $\pi^*(A)$ в таком случае будет определять выделение в процессе отдельных задач и возможный порядок их исполнения и синхронизации. В частности, множество вершин укладки $\pi^*(A)$ графа, образующих p_k-сечение (2) может быть организовано в виде отдельного потока $z_i^k(t)$ процесса $\eta_i(t)$, т.е. $V(p_k) \to z_i^k(t)$.

Пример выполнения алгоритма. Результаты применения алгоритма для произвольного графа последовательной программы (рисунок 1, таблица 1) представлены на рисунке 2, где изображена укладка графа последовательной программы, позволяющая её параллельное исполнение на вычислительной системе с четырьмя процессорными ядрами.

Таблица 1 – Параметры графа программы

v_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\tau(v_i)$	1	1	4	1	2	1	3	2	1	2	2	2	3	2	1
$E(v_i)$	0	1	3	1	1	7	2	2	4	3	8	5	5	8	10
$R'(v_i)$	0	5	0	0	0	0	3	0	0	0	0	1	2	0	0

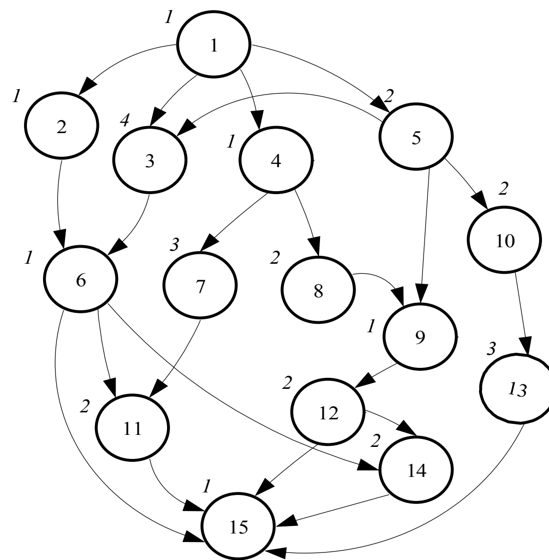

Рисунок 1 – Граф последовательной программы

Результаты численных экспериментов. Описанный алгоритм реализован на языке программирования C++. Проведены численные эксперименты с целью проверки эффективности предложенного алгоритма. В качестве исходных данных применялась последовательность произвольно-сгенерированных ациклических графов со случайными весами вершин. Анализ результатов выполнения алгоритма показал хорошую сходимость и быстродействие, соответствующее поставленным задачам. Сравнительные характе-

ристики предлагаемого алгоритма и реализации классического алгоритма поиска решения методом ветвей и границ [9] при равных экспериментальных условиях представлены на рисунке 3.

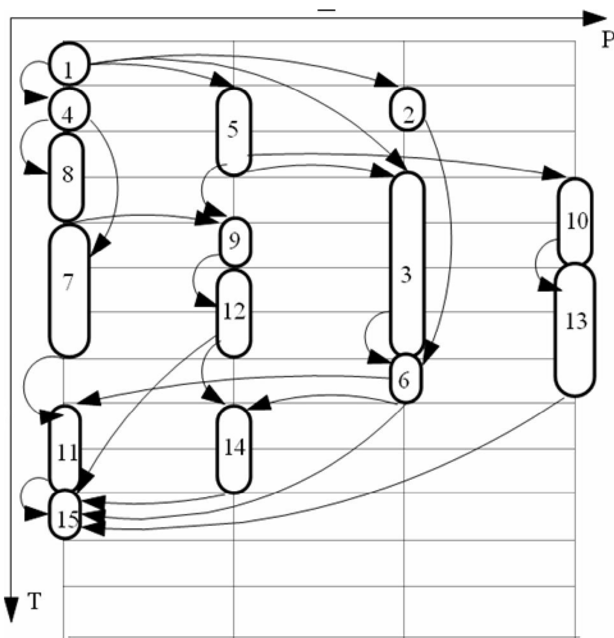

Рисунок 2 – Укладка графа программы для вычислительной системы с четырьмя процессорными ядрами

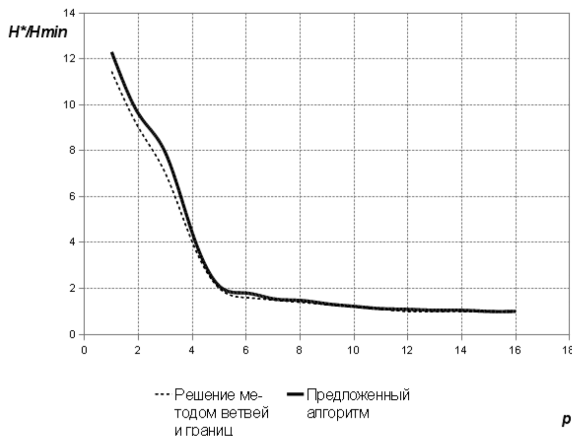

Рисунок 3 – Результаты экспериментальной оценки близости решений предлагаемого и сопоставляемого с ним алгоритма при разных значениях *p*

При этом среднее отклонение результатов выполнения алгоритма от сопоставляемого алгоритма не превышает 5-7 % и падает с ростом *p*. Трудоемкость предложенного алгоритма составляет порядка $O(\frac{N}{p})$, а сопоставляемого — порядка $O(N^3)$. Представленные результаты позволяют заключить, что предлагаемый алгоритм решает поставленные задачи.

Заключение. Реализация описанных процедур автоматизированного распараллеливания может рассматриваться как часть организации параллельной виртуальной машины и позволит приступить к решению проблемы переноса последовательных программ в среду современных многоядерных процессоров для их эффективного выполнения.

Дальнейшее повышение производительности многоядерных вычислений может быть связано с переходом от статического распараллеливания, накладывающего существенные ограничения как на возможную структуру графа последовательной программы, так и на параметры среды исполнения, к динамическому планированию. Одним из эффективных методов реализации такого планирования для многоядерных архитектур является спекулятивная многопоточность [15]. Изучение этой проблемы является предметом дальнейших исследований автора.

Библиографический список

1. *Sutter H.* The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, March 2005.

2. *Krste Asanovic* et al. The Landscape of Parallel Computing Research: A View from Berkeley. University of California, Berkeley. Technical Report No. UCB/EECS-2006-183. December 18, 2006.

3. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. – СПб.: БХВ-Петербург,2004.

4. *Крюков В.А., Удовиченко Р.В.* Отладка DVM-программ//Программирование. 2001, № 3. С. 19-29.

5. *Сальников А.Н., Сазонов А.Н., Карев М.В.* Прототип системы разработки приложений и автоматического распараллеливания программ для гетерогенных многопроцессорных систем// Вопросы атомной науки и техники, Российский федеральный ядерный центр - ВНИИЭФ, научно-технический сборник, выпуск №1, 2003 , С. 61-68.

6. *Абрамов С.М., Кузнецов А.А., Роганов В.А.* Кроссплатформенная версия Т-системы с открытой архитектурой// Вычислительные методы и программирование. 2007. Т. 8. №1. - Раздел 2. С. 175-180.

7. *Bender M.A., Rabin M.O.* Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds. Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures, July, 2000.

8. *Гришунин М.* Практика виртуализации// Открытые системы. 2009. №4.

9. *Коффман Э.Г.* Теория расписаний и вычислительные машины. М.: Наука, 1984.

10. *Мезенцев Ю. А.* Алгоритмы синтеза расписаний многостадийных обслуживающих систем в

календарном планировании// Омский научный вестник. 2006. №3(36). С.141-145.

11. *Guy Blelloch, Phil Gibbons, Yossi Matias.* Provably efficient scheduling for languages with fine-grained parallelism. Journal of the ACM, 46(2), P.281–321, 1999.

12. *Корячко В.П.* Микропроцессоры и микроЭВМ в радиоэлектронных средствах. М.: Высшая школа. 1990. 407 с.

13. *Касьянов В.Н.* Теоретико-графовые задачи анализа управляющих графов транслируемых программ// Исследования по прикладной теории графов. – Новосибирск: Наука. Сиб. Отд., 1986. – С.9–25.

14. *Корячко В.П., Скворцов С.В., Телков И.А.* Архитектуры многопроцессорных систем и параллельные вычисления: учеб. пособие.- М.: Высш.шк., 1999.

15. *Steffan J.G., Mowry T.C.* The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. Carnegie Mellon University, HPCA-4, February 1-4, 1998.