УДК: 004.272.2, 519.685, 519.686.2

Р.И. ЛЕВЧЕНКО, А.А. СУДАКОВ, С.Д. ПОГОРЕЛЫЙ, Ю.В. БОЙКО

ОБЪЕДИНЕНИЕ ПРЕИМУЩЕСТВ ПАССИВНОЙ И АКТИВНОЙ БАЛАНСИРОВКИ НАГРУЗКИ В РАМКАХ КОМПЛЕКСНОЙ СИСТЕМЫ ПЛАНИРОВАНИЯ ДЛЯ ДИНАМИЧЕСКИ РАСПАРАЛЛЕЛИВАЕМЫХ ПРОГРАММ

Анотація. *У роботі досліджуються проблеми балансування навантаження для обчислювальних ресурсів при динамічному розпаралелюванні програм у багатопроцесорних комп'ютерних системах зі слабким зв'язком. Пропонується підхід до динамічного формування графів паралельних програм, і на базі цього підходу розглядається проблема об'єднання двох типів планувальників. Обговорюються основні принципи, які дозволяють об'єднати алгоритми активного та пасивного балансування навантаження обчислювальних ресурсів, і висуваються методи розподілення завдань між цими методами балансування.*
Ключові слова: *динамічне розпаралелювання, розподілена пам'ять, кластер, балансування навантаження.*

Аннотация. *В работе исследуется проблема балансировки нагрузки для вычислительных ресурсов при динамическом распараллеливании программ на многопроцессорных компьютерных системах со слабой связью. Предлагается подход для динамического формирования графов параллельных программ, и на базе этого подхода рассматривается проблема объединения двух типов планировщиков. Обсуждаются основные принципы, позволяющие добиться объединения алгоритмов активной и пассивной балансировки нагрузки для вычислительных ресурсов, и выдвигаются методы распределения задач между рассматриваемыми методами балансировки.*
Ключевые слова: *динамическое распараллеливание, распределенная память, кластер, балансировка нагрузки.*

Abstract. *In the present work the problem of load balancing for computing resources at dynamic parallelizing of calculations for multiprocessor computer systems with loose connection is researched. The approach for dynamic creation of graphs of the parallel programs is offered, and on the basis of this approach the problem of two schedulers' types combining is considered. Main principles allowing achieving association of active and passive balancing algorithms are discussed, and methods of tasks allocation balancing for considered methods are put forward.*
Key words: *dynamic parallelizing, distributed memory, cluster, load balancing.*

1. Введение

В работе рассматриваются проблемы распределенного планирования и балансировки вычислительной нагрузки в многопроцессорных компьютерных системах со слабой связью. Такое направление исследований свойственно в первую очередь для систем динамического распараллеливания вычислений, а также для систем, реализующих алгоритмы динамической балансировки нагрузки в реальном времени. Все существующие сегодня решения в рассматриваемом направлении используют пассивную или активную балансировку нагрузки. Такие системы, как MOSIX [1], используют алгоритмы активной балансировки нагрузки для параллельных программ, написанных на основе разных транспортных библиотек, к примеру, MPI [2] или PVM [3]. Представителями пассивной балансировки нагрузки являются T-Система [4], PARUS [5] и др.

Оба рассматриваемых подхода балансировки нагрузки для многопроцессорных компьютерных систем не лишены недостатков. Активное планирование, как правило, тратит много ресурсов на постоянный мониторинг процесса распараллеливания, и не всегда такие затраты могут быть оправданными. Пассивная балансировка нагрузки часто может вообще не учитывать постоянные изменения производительности вычислительных станций, к примеру, PARUS. Также большинство систем с пассивной балансировкой не имеют

механизмов для активной балансировки нагрузки в случае сильных перепадов вычислительной мощности на компонентах распределенной системы. Для понимания проблемы стоит сказать, что система PARUS реализует статическое распараллеливание вычислений с учетом характеристик многопроцессорной системы, а Т-Система реализует автоматическое динамическое распараллеливание вычислений без поддержки миграции задач.

Целью данной работы является объединение преимуществ двух подходов активной и пассивной балансировки нагрузки для возможности построения нового планировщика в рамках авторской системы DDCI [6], реализующей автоматическое динамическое распараллеливание вычислений.

2. Требования к оформлению интерфейса для распараллеливаемых последовательных программ

Система DDCI, реализующая подход прозрачного автоматического динамического распараллеливания вычислений для блочных последовательных алгоритмов, уже зарекомендовала себя как эффективное и удобное решение для автоматического преобразования последовательных алгоритмов в параллельные аналоги [7]. Ограничения системы, связанные с распараллеливанием только блочных последовательных алгоритмов, являются обоснованными и не представляют неудобств для программирования, что было доказано в теоретических [8] и экспериментальных [9] исследованиях. В разрабатываемой системе процесс динамического построения графа распараллеливаемой программы в главной степени зависит от выбранного метода интегрирования DDCI-системы в код последовательных программ [10]. Предложенный метод интеграции DDCI-системы имеет много общего с подходом, используемым в Т-Системе [11], но те отличия, которые присутствуют, вносят серьезные изменения в процессы динамического построения и распараллеливания графа последовательной программы.

Начнем рассмотрение проблемы динамического построения графа последовательной программы с выбора задачи, на которой наглядно можно будет продемонстрировать все выдвигаемые методы. Блочный алгоритм произведения двух квадратных матриц:

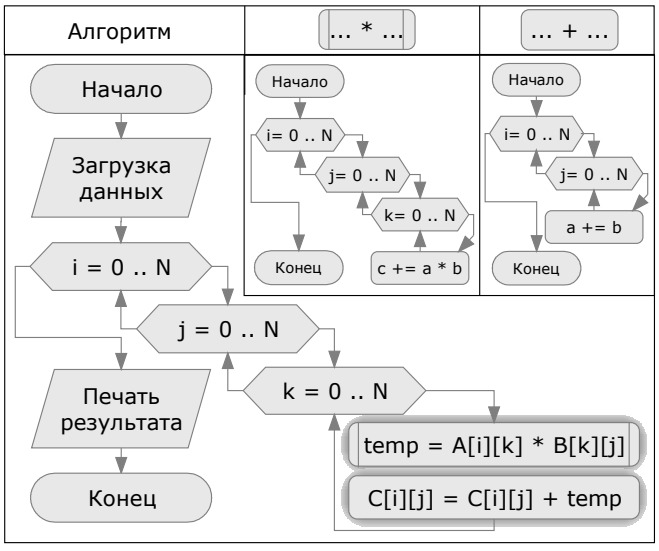

Рис. 1. Блок-схема блочного алгоритма произведения двух квадратных матриц

- имеет полиномиальную сложность задачи $O(n^3)$ в зависимости от размерности матриц;
- для задачи часто приходится использовать параллельную реализацию [12];
- блок-схему алгоритма легко анализировать (рис. 1).

На рис. 1 демонстрируются три блок-схемы, суммарно реализующие алгоритм произведения двух матриц. Основная блок-схема программы помечена как «Алгоритм». Две малых блок-схемы из левой части рисунка реализуют алгоритмы функций, отвечающих за произведение и сложение двух матриц. Вызовы данных функций можно увидеть в основной блок-схеме в правом нижнем углу. Такое разбиение алгоритма на три блок-схемы не случайно, каждая часть алгоритма имеет свои уникальные особенности.

«Алгоритм», в дальнейшем «управляющий код». Данная часть алгоритма в последовательной реализации задачи отвечает в первую очередь за связь исполняемой программы с различными локальными хранилищами данных, к примеру: параметры запуска программы, доступ к консоли, доступ к файловой системе, сетевые протоколы, специализированное исследовательское оборудование. Следствием является невозможность выполнения некоторых команд рассматриваемой части алгоритма на удаленных вычислительных станциях, а, значит, и невозможность неконтролируемого автоматического распараллеливания программы.

Второй отличительной особенностью рассматриваемой части алгоритма является потребляемая вычислительная мощность. Если в блок-схеме вызовы расчетных функций «Умножение» и «Сложение» заменить заглушками, то станет видно, что в независимости от размерности матриц эта часть алгоритма всегда выполняется одинаково быстро и не вносит ощутимой прибавки ко времени работы алгоритма в целом. Такие особенности по данному конкретному примеру означают только одно: распараллеливание управляющей части кода не является первоочередной задачей, так как это не принесет ощутимого прироста в скорости работы параллельного аналога. Теперь можно сделать некоторые промежуточные выводы:

1. Автоматическое распараллеливание рассматриваемой части алгоритма затруднено из-за проблем, связанных с локальными дескрипторами ресурсов, используемыми для связи распараллеливаемой последовательной программы с внешними хранилищами данных.

2. Существует большое количество разновидностей локальных дескрипторов, и даже если большинство из них являются стандартными, то всегда остается возможность установки на компьютере специализированного программного обеспечения или исследовательского оборудования, использующего свои «уникальные» дескрипторы. В результате детектирование локальных дескрипторов, не вводя в интерфейс распараллеливающей системы специальных директив, невозможно.

3. Если все основные вычисления распараллеливаемого блочного алгоритма находятся в чистых функциях, то вызовы функций можно заменить «заглушками», а автоматическое распараллеливание этой части алгоритма является необязательным.

4. Требование вынесения основных вычислений из основного графа программы в отдельные чистые функции есть не более чем интерфейсное решение, а, значит, достижимо для подавляющего числа блочных последовательных алгоритмов, подлежащих автоматическому распараллеливанию.

Результатом сделанных выводов является формулирование первого требования для последовательных алгоритмов: все автоматически динамически распараллеливаемые программы должны использовать основную часть графа алгоритма для связи с внешними источниками данных и использовать минимум вычислений, не инкапсулированных в отдельные чистые функции.

«Умножение», в дальнейшем «расчетные функции». В предложенной задаче многократное вычисление функции, отвечающей за перемножение двух блоков от первоначальных матриц, занимает порядка 99% времени работы всего алгоритма. В дальнейшем, при рассмотрении расчетных функций, будет подразумеваться множество помеченных чистых функций из последовательного блочного алгоритма любой задачи, которые выполняют все основные вычисления, и при этом их выполнение занимает порядка 99% времени работы всей задачи. Для всех расчетных функций в пределах реализуемого подхода также выдвигаются определенные требования, и их полное изложение уже приводилось в предыдущих работах авторов [8]. Кратко можно сказать, что рассматриваемая операция «перемножения блоков матриц» полностью отвечает всем выдвинутым ранее требованиям:

1. Учтены все ограничения для «чистой функции», согласно которым функция может работать только с теми данными, которые были переданы в качестве параметров.

2. Функция отвечает всем нормам по «эффективности чистых функций», так как количество передаваемых данных в функцию растет квадратично, а количество вычислений – кубически относительно стороны перемножаемых квадратных матриц.

В заключение можно сказать, что рассматриваемый класс функций больше всего нуждается в эффективном распараллеливании, так как суммарное время работы расчетных функций – это основная часть любой распараллеливаемой задачи.

«Сложение», в дальнейшем связующие функции. Такие функции в предыдущих авторских работах рассматривались как частные случаи редко используемых низкоэффективных чистых функций. На решение выделить их в отдельный класс повлияли следующие моменты:

1. Такие функции всегда имеют низкую эффективность, которая приводит к замедлению работы распараллеливаемой программы, причем утверждение верно, только если считать их расчетными функциями и использовать для них такие же методы планирования.

2. Так как сложность используемого в них алгоритма близка к линейной относительно количества передаваемых данных, то нет явной необходимости вычислять их всего один раз и в дальнейшем использовать результаты их работы. Более эффективным подходом для них можно считать многократный пересчет на разных станциях, так как это не добавит существенных изменений к общему времени работы всего алгоритма.

3. Отказ от учета таких функций как полноправных вершин графа распараллеливаемой части программы экономит количество используемой распределенной памяти для хранения промежуточных расчетов. Уменьшение количества распределенной памяти, в свою очередь, приводит к понижению количества межпроцессорного трафика.

В результате можно сделать следующий вывод: выделение класса связующих функций может повысить эффективность работы автоматически распараллеливаемых программ при условии, что планировщик будет иметь отдельные методы планирования для выполнения функций данного типа. Также это приводит к дополнительным интерфейсным требованиям, которые заключаются в самостоятельной пометке программистом всех связующих функций в его последовательной программе, так как система автоматического распараллеливания самостоятельно не сможет отличить связующие и расчетные функции.

3. Метод преобразования последовательных алгоритмов в параллельные

Процесс динамического построения распараллеливаемой части графа последовательной программы во многом связан с интерфейсными требованиями, которые выдвигаются системой автоматического распараллеливания. Фактически в пределах разрабатываемой DDCI-системы процесс построения графа осуществляется с помощью замен в управляющей части кода всех помеченных функций и переменных специальными «заглушками». Целью таких «заглушек» является замена всех вызовов расчетных и связующих функций на функции редактирования графа программы. В результате проводимых изменений управляющий код теряет возможность прямого доступа ко всем помеченным элементам последовательной программы, такое решение приводит к расколу всей программы на две части: управляющая и распараллеливаемая части.

Рассмотрим процесс динамического построения графа распараллеливаемой части программы более подробно на примере алгоритма блочного произведения матриц (рис. 2). На рисунке в столбце «Блок-схема алгоритма» демонстрируется блочный алгоритм произведения двух матриц, а в столбце «Процесс построения графа» демонстрируются автоматически генерируемые изменения алгоритма, которые являются неявными [13] и вносятся

через добавление специальных «заглушек» в управляющий код. Рассмотрим предлагаемые изменения:

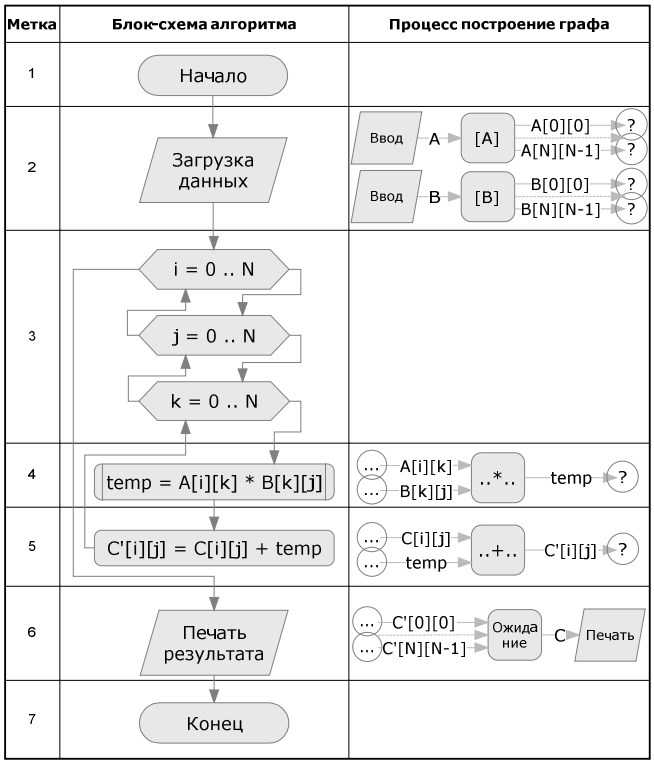

Рис. 2. Схема-процесс динамического построения графа для распараллеливаемой части блочного алгоритма произведения двух матриц

1. «Метка-2». На этапе загрузки начальных данных для решаемой задачи производится инициализация значений некоторых помеченных переменных. Изменения, вносящиеся системой распараллеливания, следующие:

• все выделения памяти на данном этапе для помеченных переменных игнорируются, вместо этого система отражает такие действия в графе программы через добавление вершин и помечает вершины как уже посчитанные;

• все операции копирования данных в помеченные переменные производятся так же, как и раньше, но по окончании операций копирования в граф программы вносятся соответствующие пометки;

• все операции копирования данных между помеченными переменными игнорируются, вместо этого вносятся дополнительные ребра в граф программы.

2. «Метка-4». Вызов функции для расчета произведения двух матриц в управляющем коде подменяется на вызов автоматически сгенерированной функции, которая добавляет соответствующую вершину в граф программы. При этом сразу же создаются все необходимые ребра графа, связывающие добавленную вершину с другими. Каждое ребро отражает направленную передачу данных из уже существующей вершины графа в создаваемую. Направление передачи данных задается программистом с помощью интерфейсных методов [8].

3. «Метка-5». Механизм установки «заглушки» для связующих функций полностью совпадает с описанным выше механизмом для расчетных функций. Разница состоит только в пометке создаваемой вершины графа как «связывающей функции» в отличие от предыдущего случая.

4. «Метка-6». Последний тип специальных «заглушек». Они используются в местах неявного преобразования типов или копирования данных из помеченных переменных в непомеченные. Таких этапов в отличие от предлагаемого примера в задаче может быть несколько. Фактически такой этап означает переход данных из области распараллеливаемой части программы в область управляющего кода. Причем управляющая часть кода изначально может работать только с явно заданными данными. «Заглушка» данного типа реализует следующие действия:

• заглушка уведомляет планировщик об ожидании данных, которые являются необходимыми для продолжения работы управляющий части кода;

• если к моменту выполнения данной операции требующиеся данные еще не доставлены на вычислительную станцию, где запущен управляющий код, то выполнение кода приостанавливается до момента прибытия необходимых данных;

• копирует данные из хранилища распределенных данных в память управляющего кода.

4. Управление динамическим графом распараллеливаемой части программы

Рассмотрим процесс формирования графа распараллеливаемой части последовательного алгоритма со стороны планировщика. Планировщик устроен так, что он является независимой системой и связан с управляющей частью кода через механизмы формирования и модификации динамического графа. Как говорилось ранее, управляющий код отвечает за процесс динамического добавления новых элементов в граф программы и при необходимости может ожидать окончания выполнения заданной вершины графа. Планирующая система отвечает за запуск чистых функций (вершин графа) на доступных вычислительных станциях, а также за последующее удаление уже посчитанных элементов из графа. При таком подходе динамически строящийся граф имеет много схожего с буфером обратного магазинного типа (FIFO). Разница состоит только в том, что планирующая система не обязана выбирать элементы динамического графа в том же порядке, в котором они были добавлены в граф. При этом практическое использование системы показало, что для большинства блочных алгоритмов минимизация межпроцессорного трафика и динамическая балансировка нагрузки приводят к тому, что планировщик подает на выполнение чистые функции примерно в таком же порядке, в каком они добавлялись в динамический граф.

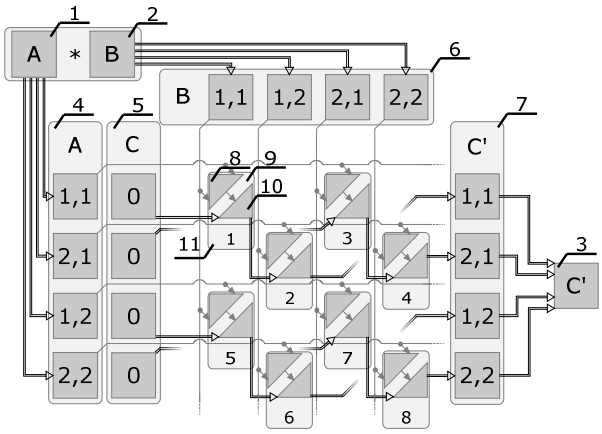

Рис. 3. Граф распараллеливаемой части блочного алгоритма произведения двух матриц

Теперь рассмотрим граф параллельной части алгоритма, который реализует блочное произведение двух квадратных матриц. Для случая разбиения каждой матрицы на 4 одинаковых квадратных блока граф будет выглядеть следующим образом (рис. 3).

В рассматриваемом примере элементы, помеченные цифрами «1» и «2», это две первоначальные перемножаемые матрицы, обе находятся в области управляющего потока. Разбиение данных матриц на блоки и переход в помеченные состояния «4» и «6» осуществляются на этапе блок-схемы «Метка-2». Процесс разбиения первоначальных матриц на блоки и копирование памяти происходят за счет процессорного времени управляющего потока. Набор обнуленных матриц под номером «5» – это набор блоков, отражающих первоначальное состояние результирующей матрицы.

Центральная часть графа, состоящая из однотипных элементов под номером «9», формируется в процессе выполнения трех циклов с «Меткой-3» в блок-схеме алгоритма. Пометки «8» и «10» соответствуют функциям перемножения и сложения двух квадратных блоков матриц соответственно. Порядок, в котором элементы будут добавляться в динамический граф программы, также показан на схеме (нумерация с пометкой «11»). Из порядка добавления элементов в динамический граф сразу видна серьезная проблема, с которой сталкивается планирующая система. Проблема заключается в том, что проведение какого-либо эффективного распределения вычислений для всех расчетных станций во время интенсивного формирования графа программы бессмысленно, так как впоследствии добавление новых элементов в динамический граф программы приведет к несовместимости конечного графа с запланированными ранее операциями. В результате данной проблемы

большинство современных систем динамического распараллеливания используют для планирования только пассивные алгоритмы анализа графа. Такие алгоритмы проводят упрощенную оптимизацию распределения вычислений для расчетных станций, так как проводить качественный анализ по распараллеливанию бессмысленно из-за быстро меняющейся формы графа. Одной из таких систем является Т-Система [4].

Рассмотрим оставшуюся часть графа распараллеливаемой части программы. Набор элементов под номером «7» соответствует результатам перемножения двух матриц. Здесь результат представлен в виде набора блоков, из которых состоит результирующая матрица. Все четыре блока к окончанию вычислений будут находиться в области распределенной памяти на разных вычислительных станциях. Объединение их в цельную матрицу под номером «3» производится на этапе блок-схемы с «Меткой-6». Процесс переноса данных из распределенного хранилища в область управляющего кода, как говорилось выше, включает в себя приостановку управляющего кода на длительный промежуток времени. Именно данную «приостановку» и предлагается использовать для запуска активных алгоритмов распараллеливания, так как именно в данный промежуток времени за изменения динамического графа отвечает только планировщик.

5. Планировщик

В предыдущем разделе было продемонстрировано, как на основе системы динамического распараллеливания вычислений можно использовать пассивные и активные алгоритмы распараллеливания в пределах одной задачи. Теперь рассмотрим проблему одновременного использования обоих подходов с целью улучшения эффективности работы комплексного планировщика для динамического распараллеливания вычислений.

Выделим два основных вида ресурсов, за эффективное использование которых отвечает планировщик:

• вычислительные ресурсы включают в себя всю вычислительную мощность используемой многопроцессорной системы;

• транспортные ресурсы включают в себя возможности транспортной системы, используемой для передачи данных между вычислительными станциями.

Теперь с целью эффективного управления рассмотренными ресурсами предлагается создать несколько очередей с разным приоритетом для возможности эффективного распределения всех имеющихся ресурсов и минимизации простоев.

1. Вычислительные ресурсы:

• высокий приоритет – используется для всех задач, запланированных активным планировщиком. Это необходимо для того, чтобы активный планировщик в случае недостачи свободных вычислительных станций мог вытеснять задачи пассивного планировщика;

• средний приоритет – используется для задач, запланированных пассивным планировщиком. Очереди высокого и среднего приоритетов используют один и тот же вычислительный ресурс, и если очередь высокого приоритета непустая, то все задания из очереди среднего приоритета приостанавливаются или могут быть отменены и перенесены на другие вычислительные станции.

2. Транспортные ресурсы:

• высокий приоритет – используется для активного планировщика, данная очередь вытесняет все последующие очереди;

• средний приоритет – используется для пассивного планировщика, а также для вытеснения очереди низкого приоритета;

• низкий приоритет – очередь используется для фонового алгоритма дублирования распределенных данных. Очередь активна во время простоя каналов, когда очереди высокого и среднего приоритетов пусты.

Для эффективного использования всех ресурсов также предлагается разбить планировщик на три независимые системы: активный планировщик, пассивный планировщик и система фонового дублирования данных.

Активный планировщик. Используется только во время приостановки управляющего кода, так как только в этот момент времени можно проводить сложный анализ графа распараллеливаемой программы, не опасаясь за серьезные и непредвидимые последующие изменения динамического графа со стороны управляющего кода. При этом во время своей работы алгоритм может использовать простаивающие вычислительные ресурсы, выделенные ранее для выполнения управляющего кода. Предлагается использовать данный планировщик для эффективного распараллеливания только той части динамического графа, результаты расчета которой ожидает управляющий поток, а остальная часть графа будет попрежнему распараллеливаться пассивным планировщиком. Такой подход направлен на минимизацию времени простоя управляющего кода и, следовательно, на минимизацию времени работы параллельной реализации программы в целом.

Пассивный планировщик. Используется постоянно и не использует сложные алгоритмы анализа графа. Предлагается для реализации планировщика использовать основные идеи Т-Системы: планировщик должен следить, чтобы как можно меньше вычислительных ресурсов простаивало[14]. При этом, в отличие от Т-Системы, предлагается ввести критерий, который будет отражать, какое количество последующих расчетов зависит от результатов выполнения каждой отдельной вершины графа. Критерий предлагается использовать для того, чтобы улучшить изначальную идею Т-Системы за счет того, что на выполнение сначала будут ставиться те вершины графа, результатов выполнения которых ожидает большее число последующих чистых функций. Такое нововведение позволит заставить пассивный планировщик в первую очередь следовать по пути минимизации диагонали графа, что, в свою очередь, приведет к повышению эффективности распараллеливания.

Система фонового дублирования данных для распределенного хранилища. Целью алгоритма является управляемое дублирование промежуточных данных между вычислительными станциями. Такой подход, во-первых, повышает вероятность того, что большая часть данных, необходимая для запуска чистой функции, заранее будет находиться на нужной вычислительной станции. Во-вторых, при запланированной передаче данных между вычислительными станциями появляется возможность выбрать для передачи недостающих данных вычислительную станцию с наиболее низкой загрузкой транспортного канала на данный момент, а не передавать данные с той единственной станции, на которой данные были посчитаны, без возможности адекватной балансировки нагрузки на транспортные каналы. При этом такой алгоритм не мешает работе пассивного и активного планировщиков, так как использует только простаивающие транспортные ресурсы.

6. Выводы

Предложен новый подход к построению системы динамического распараллеливания вычислений, позволяющий использовать не только классическое для динамических систем пассивное распараллеливание вычислений, но и активное планирование. Сформулированы интерфейсные требования для распараллеливаемых последовательных программ, а также методика преобразования последовательного алгоритма в параллельный аналог.

Определены критерии для разбиения последовательных алгоритмов на блоки, нуждающиеся в разной политике последующего распараллеливания. Разбиение предложено с целью облегчения создания системы динамического построения и управления графом распараллеливаемой программы.

Найдено решение проблемы объединения основных преимуществ активного и пассивного распараллеливания вычислений за счет разграничения областей применения обоих подходов в пределах одной задачи.

Предложена организационная структура нового комплексного планировщика для систем с динамической балансировкой нагрузки, которая включает в себя подходы активного и пассивного планирования, а также систему фонового дублирования данных.

СПИСОК ЛИТЕРАТУРЫ

1. Barak A. Scalable Cluster Computing with MOSIX for LINUX / A. Barak, O. La'adan, A. Shiloh // Proc. Linux Expo '99. – Raleigh, N.C., 1999. – May. – P. 95 – 100.
2. Jeffrey M. Squyres The Design and Evolution of the MPI-2 C++ Interface / Jeffrey M. Squyres, B. Saphir, A. Lumsdaine // Proc. Internantional Conference on Scientific Computing in Object-Oriented Parallel Computing, Lecture Notes in Computer Science. – Springer-Verlag, 1997. – P. 57 – 64.
3. New Features of PVM 3.4 and Beyond / G. Geist, J. Kohl, R. Manchel [et al.] // PVM Euro Users' Group Meeting. – Lyon, France, Hermes Publishing, Paris, 1995. – September. – P. 1 – 10.
4. Nesterov I.A. Towards programming of numerical problems within the system providing automatic parallelizing / I.A. Nesterov, I.V. Suslov // Proc. of 7th SIAM conference on parallel processing for scientific computing. – San-Francisco, CA, 1995. – P. 716.
5. Alexey N. Salnikov PARUS: A Parallel Programming Framework for Heterogeneous Multiprocessor Systems / N. Alexey // Lecture Notes in Computer Science. Recent Advantages in Parallel Virtual Machine and Message Passing Interface. – 2006. – Vol. 4192. – P. 408 – 409.
6. Система автоматического динамического распараллеливания вычислений для многопроцессорных компьютерных систем со слабой связью (DDCI) / Р.И. Левченко, А.А. Судаков, С.Д. Погорелый [и др.] // Управляющие системы и машины. – 2008. – № 3. – С. 66 – 72.
7. Levchenko R.I. Parallel software for modeling complex dynamics of large neuronal networks / Levchenko R.I., Sudakov O.O., Maistrenko Yu.L. // Proc. 17th International Workshop on Nonlinear Dynamics of Electronic Systems, (June 21–24, 2009). – Rapperswil (Switzerland), 2009. – P. 34 – 37.
8. Левченко Р.И. Проблемы эффективности автоматического динамического распараллеливания вычислений для многопроцессорных систем со слабой связью / Р.И. Левченко, А.А. Судаков, С.Д. Погорелый [и др.] // Проблеми програмування. – 2010. – № 2–3. – Спецвипуск. Матеріали міжнар. конф. УкрПРОГ 2010. – С. 178 – 184.
9. Levchenko R.I. DDCI: Dynamic parallelizing system for high performance computing clusters / R.I., Levchenko, O.O. Sudakov // Proc. of the seventh international young scientists' conference on applied physics, (June 13–15, 2007). – Kyiv (Ukraine), 2007. – P. 147 – 148.
10. Levchenko R.I. DDCI: Simple Dynamic Semiautomatic Parallelizing for Heterogeneous Multicomputer Systems / R.I. Levchenko, O.O. Sudakov, S.D. Pogorelij // Proc. of the 5-th IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, (September 21–23, 2009). – Cosenza (Italy), 2009. – P. 208 – 211.
11. Adamovich A.I. CT: an Imperative Language with Parallelizing Features Supporting the Computation Model "Autotransformation of the Evaluation Network" / A.I. Adamovich // Proc. LNCS #964 1995, Parallel computing technologies: third international conference. PaCT-95, (September 12–25, 1995). – St.-Petersburg, Russia, 1995. – P. 127 – 141.
12. Keren A. Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster / A. Keren, A. Barak // IEEE Tran. Parallel and Distributed Systems. – 2003. – Vol. 14, Is. 1. – P. 39 – 50.
13. Levchenko R.I. DDCI: Interface for Transparent parallelizing of calculations / R.I. Levchenko, O.O. Sudakov, S.D. Pogorelij // Proc. of the ninth international young scientists' conference on applied physics, (June 17–20, 2009). – Kyiv (Ukraine), 2009. – P. 12.
14. Abramov S. T-system: programming environment providing automatic dynamic parallelizing on IP-network of Unix-computers [Електронний ресурс] / S. Abramov, A. Adamovitch, M. Kovalenko // Report on 4-th International Russian-Indian seminar and exhibition. – Moscow, 1997. – Sept. 15–25. – Режим доступу: http://www.botik.ru/~abram/ts-pubs.ht.

Стаття надійшла до редакції 31.03.2010