УДК 519.688

ОБ ОДНОМ АЛГОРИТМЕ УПРАВЛЕНИЯ ПАРАЛЛЕЛЬНЫМИ ВЫЧИСЛЕНИЯМИ С ДЕЦЕНТРАЛИЗОВАННЫМ УПРАВЛЕНИЕМ

© Е.А. Ильченко

Ключевые слова: параллельный алгоритм; децентрализованное управление; система Mathpar; SPMD вычислительная парадигма

Дается описание одного алгоритма децентрализованного управления параллельным вычислительным процессом, основанного на SPMD вычислительной парадигме.

1 Введение

Проблема управления вычислительным процессом на кластере с распределенной памятью является одной из наиболее трудных задач в теории параллельного программирования. Чем равномернее будет распределена нагрузка на узлы кластера, тем эффективнее будет его работа. Для достижения выполнения этих условий существуют методы, которые можно разбить на 2 типа: *статические и динамические.*

В *статических методах* жестко фиксирована общая схема вычислений и передачи информации, и каждый процессор в любой момент времени «знает», какое следующее действие выполнить. В *динамических методах* каждый процессор меняет план действий по ходу выполнения алгоритма. В случае если есть свободный процессор, то ему отдается часть задания, а если нет такого процессора, то она вычисляется на текущем процессоре. Статический методы обычно предпочтительнее, чем динамические в тех случаях, когда имеются регулярные плотные данные.

Можно различать два типа динамических методов: с централизованным управлением и с децентрализованным управлением.

При *централизованном управлении* динамическое распределение заданий предполагает наличие диспетчера, контролирующего вычислительный процесс. Роль диспетчера выполняет один из узлов кластера. Изначально все процессоры являются свободными и находятся в списке свободных процессоров, который хранит диспетчер. Диспетчер отвечает на 2 вида запросов:

1) предоставить некоторому узлу M свободных процессоров, при этом он их исключает из списка свободных процессоров;

2) добавить освободившиеся процессоры в имеющийся список свободных.

Другой способ динамической реализации алгоритма — создание диспетчера на каждом узле кластера. Такое управление вычислительным процессом будет *децентрализованным.* Это можно сделать, например, в двухпоточном режиме. Один поток отвечает за ход вычислительного процесса, а второй поток отвечает за пересылку задач и пересылку

списков свободных процессоров. Далее речь пойдет именно о таком способе организации параллельных вычислений. При этом мы продолжаем исследования, начатые в работах [1–13], посвященных динамическим алгоритмам управления параллельным вычислительным процессом. И непосредственно опираемся на работы [1, 8], посвященные децентрализованному динамическому управлению.

2 Приоритеты потоков

Программа, реализующая схему децентрализованного управления вычислениями, написана на языке Java с использованием пакета MPIJava. Язык Java имеет встроенные средства для работы с потоками. Каждому потоку приписан параметр, которое определяет относительный приоритет потока. Приоритеты – это целые числа от 1 до 10. Например, если 2 потока имеют приоритеты 2 и 8, то теоретически первому потоку будет отведено 20 % процессорного времени, а второму — 80 %.

Мы используем два потока — *счетный* поток и *диспетчерский* поток. В счетном потоке будет происходить решение полученной задачи, в диспетчерском — обмен свободными процессорами, получение входных данных задачи и отправка результата. Счетному потоку назначается приоритет 1, диспетчерскому — 10. Рассмотрим их работу подробнее.

Выполнение потока может быть приостановлено функцией Sleep на некоторое время и мы используем такую функцию в диспетчерском потоке. Диспетчерский поток реализован в виде бесконечного цикла, в теле которого выполняются операции по приему и отсылке задач, обмену свободными процессорами. В конце каждой итерации этого цикла он будет приостанавливаться на заданное время sleepTime, чтобы полностью передать управление счетному потоку.

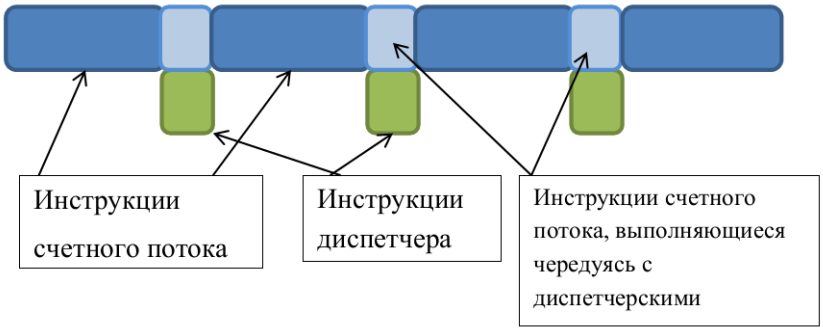

Рис. 1. Очередность выполнения инструкций счетного и диспетчерского потоков

3 Очередь задач

Счетный и диспетчерский поток работают с т. н. очередью задач. Очередь задач — это структура данных, которая поддерживает следующие 3 операции:

1) вернуть произвольную задачу из очереди по ее идентификатору. Это операция нужна исключительно счетному потоку;

2) добавить задачу в конец очереди. Эта операция также нужна только счетному потоку;

3) взять N задач из начала очереди. Этой операцией будет пользоваться диспетчерский поток для передачи этих задач другим процессорам.

Нужно заметить, что операция 1 не будет выполнена в том случае, если задача уже была отправлена другому процессору. Если в очереди имеется $K < N$ задач, то в результате выполнения операции 3 будут возвращены все K задач, другими словами, очередь станет пустой. Для обоих потоков, счетного и диспетчерского, очередь задач одна.

Очередь задач реализована в виде класса TaskQueue, имеющего следующий интерфейс:

1) `public synchronized boolean TryReturnTaskFromQueue(AbstractTask t);`

2) `public synchronized void PushTaskInQueue(AbstractTask t);`

3) `public synchronized ArrayList<AbstractTask> RemoveTasksForSending(int N);`

4 Граф алгоритма

В общем случае алгоритм может состоять из нескольких типов задач, например из таких:

- $A * B$ — матричное умножение;
- $A * B + C * D$ — сумма двух произведений матриц;
- A^{-1} — обращение матрицы.

Поскольку счетный поток работает с абстрактными классами AbstractTask и AbstractGraphOfTask, для каждого типа задачи должно быть написано по одному наследнику этих классов. Класс AbstractTask имеет следующий интерфейс:

1) `public synchronized void SetTaskCompleted();`

2) `public synchronized boolean  IsTaskCompleted();`

3) `public synchronized boolean IsTaskSendet();`

4) `public synchronized void SetTaskSendet();`

5) `public abstract void SetStartTask(String []args);`

6) `public abstract boolean IsLittleTask();`

7) `public abstract void ProcLittleTask();`

8) `public abstract void SendTaskToNode(int node);`

9) `public abstract void RecvTaskFromNode(int node);`

10) `public abstract void SendResultToNode(int node);`

11) `public abstract void GetResultFromNode(int node);`

Методы 5-11 являются абстрактными и должны быть реализованы в классе-наследнике. В методе `SetStartTask(String []args)` должна быть инициализация данных для стартовой задачи.

Метод `IsLittleTask()` должен вернуть true, если имеющуюся задачу больше нет смысла делить, иными словами, время ее обработки на текущем узле будет меньше, чем время пересылки и обработки ее на другом узле. Метод `ProcLittleTask()` должен содержать код обработки неделимой задачи. В методах 8-11 должен быть описан прием и отправка данных текущей задачи, прием и отсылка результата. Любая задача имеет некоторый граф алгоритма, дуги которого устанавливают порядок, в котором происходят вычисления в вершинах. Пусть некоторая задача T состоит из N подзадач. Пронумеруем задачи от 0 до $N-1$. Нужно построить ориентированный граф G, множеством вершин которого будут N подзадач. Множество связей определим следующим образом: из вершины q в вершину v должна быть направлена дуга, если задача v может быть решена только после задачи q. Такая зависимость появляется в том случае, когда часть входных данных для задачи v появляется в результате выполнения задачи q.

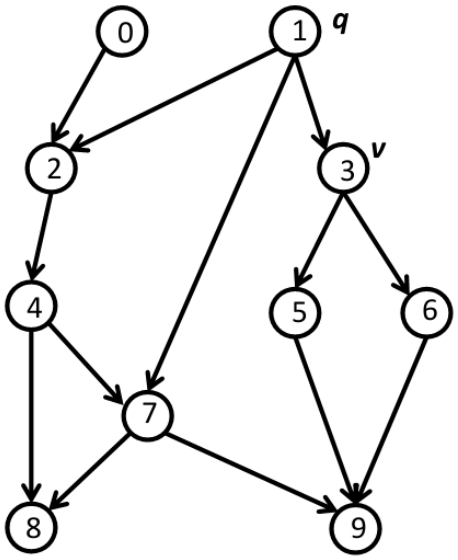

Рис. 2. Пример графа алгоритма

Класс AbstractGraphOfTask имеет следующий интерфейс:

1) `abstract public void InitVertex(`
`int numb, AbstractTask currentTask,AbstractTask []allVertex);`

2) `abstract public void FinalizeVertex(int numb, AbstractTask currentTask,`
`AbstractTask []allVertex);`

3) `abstract public void FinalizeGraph(AbstractTask currentTask,`
`AbstractTask []allVertex);`

Вершины этого графа — наследники класса AbstractTask. В первом методе должна быть подготовка данных для вычислительного блока в вершине с номером numb, во втором методе — ее постобработка, и в последнем — обработка данных счета всех вершин и вычисление результата исходной задачи.

5 Счетный поток

Счетный поток реализован в виде бесконечного цикла, в котором происходит ожидание задачи от диспетчера. Как только задача получена, поток переходит в режим счета. Алгоритм обработки графа текущей задачи выглядит следующим образом:

1. Пока вычисления во всех вершинах графа не завершены:
   1.1. Выполнить постобработку всех вершин, которые были поставлены в очередь и их вычисления были выполнены на других узлах. Пометить их как вершины с завершенными вычислениями.
   1.2. Если имеются вершины, доступные для счета $(0..k)$:
      1.2.1. Выполнить инициализацию всех доступных вершин.
      1.2.2. Вершины с 1 по k отправить в очередь задач.
      1.2.3. Выполнить задание, соответствующее вершине с номером 0, затем выполнить его постобработку.
   1.3. Если нет доступных для счета вершин:
      1.3.1. Взять из очереди произвольную вершину текущего графа и начать решение соответствующего ей задания, затем выполнить ее постобработку.
2. Выполнить постобработку всего графа.

6 Диспетчерский поток

Диспетчерский поток имеет список свободных процессоров, список дочерних процессоров, список задач, отправленных на дочерние узлы и для каждого дочернего узла хранится номер последнего отправленного ему процессора.

Диспетчерский поток реализован в виде бесконечного цикла и имеет 3 режима:

1) **0 — режим ожидания задачи**. В этом режиме ожидается задача от любого другого процессора. Может прийти одно из двух сообщений: либо задача, которую необходимо решить и результат вернуть отправителю, либо сигнал к завершению. Сигнал к завершению отсылается корневым узлом, то есть тем, на котором была запущена стартовая задача. Если получен сигнал к завершению, происходит выход из цикла. Иначе выполняются действия по приему данных задачи, затем она полностью передается счетному потоку и происходит переход в режим 1;

2) **1 — основной режим**. В этом режиме счетный поток решает имеющуюся задачу и производит обмен свободными процессорами. В этом режиме происходят следующие действия:

- получение свободных процессоров от родительского узла;
- опрос дочерних узлов о завершенности работы;
- прием данных от дочерних узлов, завершивших работу;
- создание новых дочерних узлов;

- отправка дочерним узлам имеющихся свободных процессоров;
- проверка того, что вся задача, полученная от родительского узла решена.

Опишем эти действия более подробно. На *шаге 1* проверяется, пришло ли сообщение от родительского узла со свободными процессорами. Если это произошло, то полученные номера процессоров добавляются в список свободных. На *шаге 2* опрашиваются все дочерние узлы. Если от какого-то узла пришел запрос на завершение, происходит отправка этому дочернему узлу номера последнего отосланного свободного процессора (lastSendet). Дочерний узел не вернет результат до тех пор, пока в его списке свободных узлов не окажется процессор с номером lastSendet. Это нужно для предотвращения следующей ситуации (рис. 3):

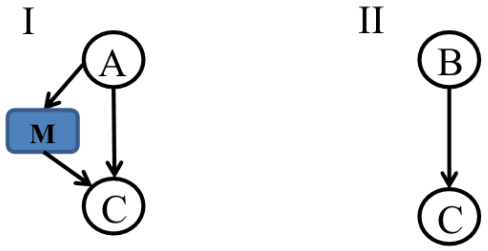

Рис. 3. Отправка сообщения от одного узла кластера другому

На шаге I для процессора с номером C родительским является процессор A. Процессор A отсылает ему сообщение M со свободными процессорами. Но до приемки этого сообщения процессор C завершил работу и перешел в режим ожидания задачи. Сообщение M со свободными процессорами принято не было, но из списка вершины A они были исключены. После этого для процессора C родительским стал другой узел с номером B. Но поскольку сообщение со свободными узлами принимается именно от текущего родительского узла, то сообщение M, возможно, никогда не будет принято, и узлы, содержащиеся в нем, будут утеряны. Для предотвращения этой и других подобных ситуаций и нужен механизм запоминания – отправки последней отосланной вершины. Если этому узлу не отправлялись свободные процессоры, то переменная lastSendet для этого узла будет иметь специальное значение: -2.

На шаге 3 для каждого дочернего узла делается проверка получения от него сообщения со свободными процессорами. Если сообщение пришло, то полученные номера заносятся в список свободных. Поскольку отправка свободных вершин от дочернего узла к родительскому осуществляется только после того, как дочерний узел завершил решение задачи, сразу происходит прием результата. Задача, отправленная этому дочернему узлу, помечается как выполненная. Номер узла переносится из списка дочерних процессоров в список свободных процессоров.

На четвертом шаге создаются новые дочерние узлы. Как один из вариантов управления параллельным вычислительным процессом, диспетчер может иметь настраиваемый параметр, который определяет максимальное количество дочерних узлов (maxD). Он может, например, экспериментально подбираться для каждого алгоритма. Но это число не может быть меньше двух. Слишком большое значение этого параметра также может приводит к ухудшению работы системы. Если число имеющихся дочерних узлов меньше maxD, будут созданы новые. Обозначим общее количество имеющихся свободных процессоров через Nfree, число дочерних узлов — через Dnodes. Допустим, имеется возможность создать еще

L дочерних узлов. L определяется как минимум из 3 величин: maxD-Dnodes, Nfree и количества задач, имеющихся в очереди. Затем произвольным образом из списка свободных процессоров берется L номеров, отсылается им ровно по одной задаче и эти процессоры помещаются в список дочерних узлов.

На пятом шаге все имеющиеся свободные процессоры делятся на Dnodes групп. Затем каждая группа отсылается соответствующему дочернему узлу.

Далее проверяется тот факт, что вся имеющаяся задача решена. Если задача решена на процессоре с номером 0, то всем узлам отсылается сигнал к завершению работы. Если же задача решена на другом узле, родительскому узлу отправляется запрос на завершение и происходит переход в режим 2;

3) **2 — режим завершения работы узла**. В этом режиме происходят следующие действия. Если пришло сообщение со свободными узлами от родительского узла, добавим их в список свободных узлов. Если был получен номер lastSendet, делается проверка того факта, что в списке свободных процессоров он присутствует. Если он действительно присутствует, то родительскому узлу отсылаются все имеющиеся свободные узлы, следом отправляется полученный результат вычислений. Если число lastSendet не было получено, проверяется, пришло ли сообщение с ним.

На рис. 4 (а-ж) показан пример того, как может происходить распределение свободных процессоров. Здесь вершины - узлы кластера, подписи над вершинами - список имеющихся свободных процессоров у данного узла кластера.

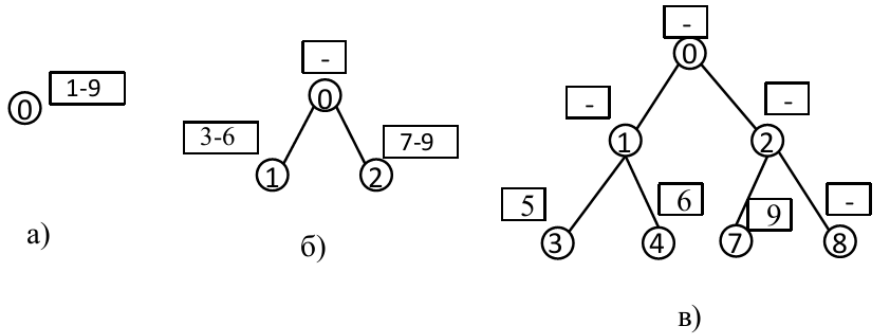

Рис. 4 (а-в). Начальная стадия построения дерева выполнения программы

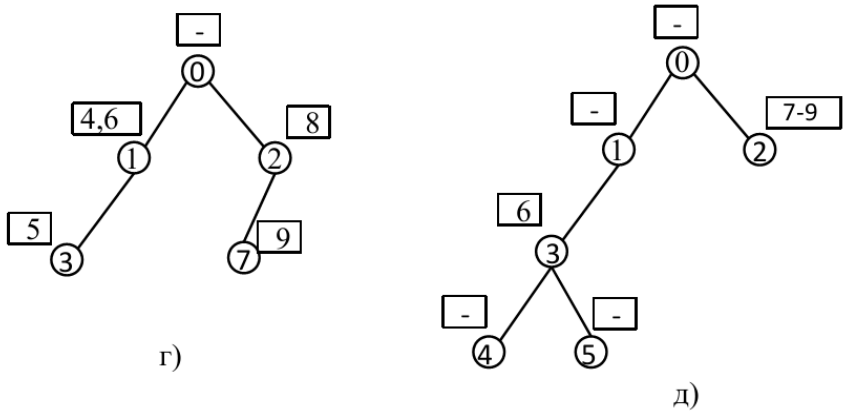

Рис. 4 (г, д). Завершение работы некоторых узлов кластера

Рис. 4 (е, ж). Завершающая стадия работы программы

7 Заключение

Описанный подход является одним из возможных вариантов организации параллельных вычислений с децентрализованным управлением. Исследование этого алгоритма находится на начальном этапе. Запланирована серия экспериментов для алгоритмов компьютерной алгебры на кластере МВС-100К Межведомственного суперкомпьютерного центра РАН.

ЛИТЕРАТУРА

1. *Malaschonok G., Ilchenko E.* Decentralized control of parallel computing // International conference Polynomial Computer Algebra. St.Petersburg: PDMI RAS, 2012. P. 57-58.

2. *Малашонок Г.И.* Конструктивная математика и принцип близкодействия // Вестник Тамбовского университета. Серия Естественные и технические науки. Тамбов, 2011. Т. 16. Вып. 1. С. 115-120.

3. *Malaschonok G.I.* Project of Parallel Computer Algebra // Tambov University Reports. Series Natural and Technical Sciences. Tambov, 2010. V. 15. Issue 6. P. 1724-1729.

4. *Бетин А.А.* Эксперименты с параллельным алгоритмом вычисления присоединенной матрицы и параллельным умножением файловых матриц // Вестник Тамбовского университета. Серия Естественные и технические науки. Тамбов, 2010. Т. 15. Вып. 1. С. 341-345.

5. *Бетин А.А.* Эксперименты с параллельным алгоритмом вычисления присоединённой матрицы // Вестник Тамбовского университета. Серия Естественные и технические науки. Тамбов, 2010. Т. 15. Вып. 6. С. 1748-1754.

6. *Малашонок Г.И.* Компьютерная математика для вычислительной сети // Вестник Тамбовского университета. Серия Естественные и технические науки. Тамбов, 2010. Т. 15. Вып. 1. С. 322-327.

7. *Malaschonok G.I.* Architecture of ParCA3 project // International conference Polynomial Computer Algebra. St.Petersburg, PDMI RAS, 2010. P. 44-47.

8. *Малашонок Г.И.* Управление параллельным вычислительным процессом // Вестник Тамбовского университета. Серия Естественные и технические науки. Тамбов, 2009. Т. 14. Вып. 1. С. 269-274.

9. *Малашонок Г.И., Валеев Ю.Д.* Организация параллельных вычислений в рекурсивных символьно-численных алгоритмах // Труды конференции ПаВТ'2008 (Санкт-Петербург). Челябинск: Изд-во ЮУрГУ, 2008. С. 153-165.

10. *Малашонок Г.И., Валеев Ю.Д.* Рекурсивное распараллеливание символьно-численных алгоритмов // Вестник Тамбовского университета. Серия Естественные и технические науки. Тамбов, 2006. Т. 11. Вып. 4. С. 536-549.

11. *Малашонок Г.И., Валеев Ю.Д.* Динамическое распределение заданий в вычислительном кластере по графу алгоритма // XI Державинские чтения. Тезисы докладов. Тамбов: Изд-во ТГУ им. Г.Р. Державина, 2006. С. 38-47.

12. *Malaschonok G.I.* In the Direction of Parallel Computer Algebra System // Computer Science and Information Technologies. Proc. Conf. (Sept.19-23, 2005. Acad. Sci. of Armenia.) Yerevan, 2005. P. 451-453.

13. *Малашонок Г.И., Валеев Ю.Д.* О некоторых подходах к построению параллельных программ // Вестник Тамбовского университета. Серия Естественные и технические науки. Тамбов, 2005. Т. 10. Вып. 1. С.154-156.

БЛАГОДАРНОСТИ: Работа выполнена при поддержке гранта РФФИ № 12-07-00755-а.

Поступила в редакцию 20 декабря 2012 г.

Ilchenko E.A. AN ALGORITHM FOR THE DECENTRALIZED CONTROL OF PARALLEL COMPUTING PROCESS.

We describe an algorithm for the decentralized control of parallel computing process which is based on the SPMD computational paradigm.

Key words: parallel algorithm, decentralized control, the system Mathpar, SPMD computational paradigm.