Разработка эффективного алгоритма поточного запуска групп вычислительных задач на кластере

Ю. А. Типикин[1,a]

[1]Санкт-Петербургский Государственный Университет,
Россия, 199034, Санкт-Петербург, Университетская наб. д.7-9.

E-mail: [a] ytipikin@spbu.ru

С ростом числа узлов в кластерах все более серьезным становится вопрос о надежности и стабильности выполняемых параллельных вычислений. Существующие решения в основном основываются на неэффективных процессах сохранения стека оперативной памяти в надежные хранилища. Создание подобных точек сохранения (checkpoints) в масштабах больших вычислительных комплексов может быть не применим вовсе. В этой работе мною была исследована модель Акторов, которая является моделью распределенных вычислений, и на ее основе мною был предложен алгоритм эффективного запуска задач на кластерах после прерывания без использования точек сохранения. Этот алгоритм является частью модели, которую я условно называю «моделью управляющих объектов» по названию ключевого компонента - управляющего ядра. В этой работе мною описаны все компоненты модели, ее внутренние процессы и проведен анализ использования модели.

Ключевые слова: распределение задач, надежность, кластер

Введение

Одним из самых известных способов восстановления является создание контрольных точек (checkpoints). Основная идея данного подхода состоит в сохранении полного состояния вычислительного узла на постоянную память, с последующим обновлением диффиренцируемых частей. Подобная операция очень затратна, так как требует копирования большого массива данных из ОЗУ на жесткий диск, при этом всегда есть опасность частичной записи данных. Тем не менее, с алгоритмической точки зрения, данный подход несложен. Есть и другие способы, менее тривиальные — приведем их классификацию.

В статье «Fault tolerance-challenges, techniques and implementation in cloud computing» [Bala, Chana, 2012] авторы делят известные алгоритмы на две группы: превентивные и поствременные.

Например, в первую группу включены такие техники: перезапуск задачи [Sindrilaru et al., 2010], перемещение задачи на другой узел, контрольные точки [Nayeem, Alam, 2006]. Во второй группе авторы отметили следующие методы: программное омоложение [Armbrust et al., 2010], когда система периодически полностью перезагружается; упреждающая миграция, когда процесс-наблюдатель может заранее выполнить миграцию задачи на другие узлы, принимая в расчет текущую статистику; самолечение узлов — все процессы запускаются в виртуальных машинах, одновременно с этим запускаются копии этих процессов. В случае необходимости они подменяются.

Основной недостаток приведенных выше решений в том, что они изначально ограничены неизменной програмной реализацией алгоритма. Чтобы обеспечить программам свойство надежности, в этом случае необходимо создавать дополнительные сопровождающие конструкции, а это, в свою очередь, ведет к увеличению накладных расходов системы. Далее в этой работе будет предложен способ, который описывает внутренний, по отношению к программе, процесс восстановления.

В качестве теоретической основы такого алгоритма восстановления была выбрана модель акторов К. Хьюита [Baker, Hewitt, 1977]. Используя базовые принципы модели акторов модифицируем ее так, чтобы стало возможным восстанавливать работу системы после сбоя. Назовем новую модель «моделью управляющих объектов». Основное отличие модификации заключается в наличии межпроцессовой иерархии.

Описание модели

Модель управляющих объектов опишем на основе определений ее компонентов.

Определение 1. Управляющий объект (ядро) — основная единица модели. Объект, который содержит информацию о своем состоянии, свои переменные и процедуры, определяющие поведение. Набор ядер формирует древовидную иерархию строго порядка. Каждое ядро в процессе выполнения может породить любое количество других ядер.

Определение 2. Ядро-потомок — порожденное ядро, содержащее независимую (параллельную) часть алгоритма.

Определение 3. Вычислительный сервер (сервер) — процесс, который распределяет ядра по локальным процессам. Каждое перемещенное ядро запускается в отдельном процессе. Так как виртуальная общая память у каждого процесса своя, сервер выполняет роль коммутатора между процессами. Он ждет поступления ядер от других серверов в кластере и сам может их отправлять, а также следит за корректной работой порожденных процессов.

Сервера также находятся в иерархии, что упрощает балансировку ядер между ними. Другой функцией сервера является распределение файлов журналов внутри кластера, которые обеспечивают процесс восстановления после сбоя.

Определение 4. Журнал операций — абстрактное представление всех состояний ядер, расположенных в порядке иерархии дерева. Журнал операций «распределен» среди серверов.

Определение 5. Журнал-на-поток — основа процесса восстановления. Представляет собой последовательно обновляемый файл на устройстве постоянной памяти, содержащий последовательные записи состояний вычислительных ядер.

Определение 6. Конвейер — совокупность очереди, состоящей из ядер одной задачи, потоков-обработчиков и журнала перемещаемых ядер. На разных серверах могут быть конвейеры, обрабатывающие одну и ту же программу, но набор ядер у них будет непересекающимся.

Достижение эффективного алгоритма запуска задачи есть преобразование исходной последовательности операций задачи во множество управляющих объектов. Такие «правила разбиения» можно сформулировать следующим образом:

1. Внутри ядра должен находится логически законченный участок исходного алгоритма. То есть, результат выполнения объединяемых операций должен выражаться в чем-то конкретном, например, в возвращаемом значении. Чем лучше мы сможем сгруппировать операции, тем меньшим по объему получится множество ядер и тем меньшими будут затраты на поддержание их жизнедеятельности. В общем случае, примером логически законченного участка алгоритма можно считать функцию. Ядро же являются расширением функции и может включать в себя как тело функции, так и ее контекст — значения переменных, зависимости, промежуточные результаты.

2. Внутри ядра все операции последовательны.

3. Параллельные участки алгоритма реализуются путем создания нескольких дочерних ядер, которые распределяются между вычислительными узлами и процессорами.

4. Каждый участок алгоритма обладает своим контекстом данных. Так, в «вычислительных» ядрах необходимо придерживаться локальной области видимости переменных и не допускать ситуации, когда переменная обрабатывается в не связанных отношением потомок-родитель ядрах.

5. Принцип локальной видимости переменных влечет необходимость дублирования переменных в каждом ядре, где они подвергаются изменению.

Если исходный алгоритм разбит на управляющие объекты в соответствии с этими правилами, тогда в любой момент вычислений состояние всей системы может быть отражено в журнале операций, а значит может быть восстановлено.

Далее опишем базовый API модели, его работу, восстановление. Каждый управляющий объект в процессе своего существования проходит 3 этапа: два из них не зависимо от того, штатно работает система или нет и еще один опциональный. Первый этап — порождающий, определяется в методе act(). Здесь программистом определяется логика поведения ядра, описывается его основное назначение. Здесь на любом этапе может происходит создание любого количества ядер-потомков. После порождения, ядра-потомки также проходят первый этап. После заверешения act() порожденного потомка, исполняется метод react() у

родителя, знаменующий завершение работы принадлежащей ядру-потомку части алгоритма. В этот же момент результат передается ядру-родителю. Последний этап опционален и выполнеяется только в случае аварии — исполняется метод rollback(). При переходе между этапами происходит запись состояния ядра в журнал.

Состояние программы целиком определяется набором ядер. Запись в журнал значений переменных, ссылок происходит постоянно, при этом файлы журналов распределяются между узлами кластера в рамках распределенной файловой системы. Журнальная запись содержит исполняемую часть алгоритма, так что в случае прерывания всегда известена следующая операция.

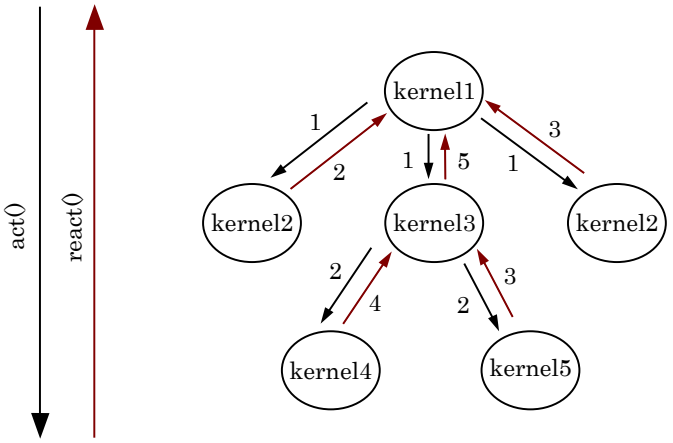

Рис. 1. Иерархия вычислительных ядер. Здесь цифрами обозначена очередность выполнения операции

В случае сбоя, основную роль играет функция rollback(). В ней описывается логика, которая очистит состояние системы, если выполнялись не идемпотентные операции. Очевидно, что для каждого вычислительно ядра данная функция уникальна. После выхода из строя одного или нескольких узлов, иерархия узлов кластера ищет записи состояний затронутых ядер на «живых» узлах. Современные распределенные файловые системы позволяют реплицировать данные с достатчной скоростью, чтобы последнее найденое корректное (с точки зрения целостности записи на диске) состояние конкретного управляющего объекта было максимально близким к моменту сбоя. Далее, из этого состояния восстанавливается ядро, которое вызывает собственный метод rollback(), после завершения метода попадает в очередь ядер «на выполнение». Ясно, что в какой-то момент узел окажется перегруженным, но включении убывших узлов, очередь оперативно переместит крайние ядра на вновь открывшийся ресурс.

Тестирование и заключение

В качестве проверки работоспособности модели, реализуем алгоритм рассчета некоторой задачи, используя управляющие объекты: посчитаем сумму ряда

$$\sum_{0}^{500000=l} \tan(x+10)\sin(x-10) + 1.34p,$$

где p некоторый коэффициент, расчитаный как

$$\sum_{0}^{5000000} 0.345\sin(x) + \cos(10x).$$

Тестовая программа содержит 10 ядер, каждое из которых считает ряды с шагом $p = \frac{l}{10}$. Ядро, в котором первоначально рассчитывается коэффициент p, будет являться корнем иерархии. Для тестирования процесса восстановления будем уменьшать число успешно завершенных ядер на единицы за один шаг и измерять время выполнения, требуемое для заверешения рассчета суммы ряда.

Рис. 2. Производительность алгоритма восстановления при различном количестве успешно завершенных ядер по сравнению с производительностью без ошибок (идеальное время).

Верхняя область графика 2 отражает общее время работы программы, требуемое на выполнение всех операций с учетом восстановления. Нижняя область соответствует времени выполнения программы до сбоя. Пунктиром обозначено время работы программы без сбоев.

Контролируемое тестирование свойств модели управляющих объектов, результат которого отражен на рис. 2, показывает, что вычислительная задача восстанавливается (т.е. успешно завершает исходную последовательность операций) не зависимо от момента прерывания. Производительность модели выше, чем при полном перезапуске программы. В худшем случае — после 5 шагов — имеем следующее отношение времени выполнения: $\frac{46}{70} = 0,657$; здесь 46 с. — итоговое время выполнения программы после восстановления, 70 с. — время выполнения программы в случае перезапуска программы после 5 выполненных шагов.

Список литературы

Armbrust M., Fox A., Griffith R. et al. A view of cloud computing // Communications of the ACM. — 2010. — Vol. 53, no. 4. — P. 50–58.

Baker H., Hewitt C. Laws for communicating parallel processes. — 1977.

Bala A., Chana I. Fault tolerance-challenges, techniques and implementation in cloud computing // IJCSI International Journal of Computer Science Issues. — 2012. — Vol. 9, no. 1. — P. 1694–0814.

Nayeem G. M., Alam M. J. Analysis of Different Software Fault Tolerance Techniques. — 2006.

Sindrilaru E., Costan A., Cristea V. Fault tolerance and recovery in grid workflow management systems // Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on / IEEE. — 2010. — P. 475–480.

Development of algorithm for efficient pipeline execution of batch jobs on computer cluster

Iu. A. Tipikin[1,a]

[1]Saint Petersburg State University,
7/9 Universitetskaya emb., St. Petersburg, 199034, Russia

E-mail: [a] ytipikin@spbu.ru

The problem of reliability and stability of high performance computing parallel jobs become more and more topical with the increasing number of cluster nodes. Existing solutions rely mainly on inefficient process of RAM dumping to stable storage. In case of really big supercomputers, such approach – making checkpoints - may be completely unacceptable. In this study, I examined the model of distributed computing – Actor model - and on this basis I developed an algorithm of batch jobs processing on a cluster that restores interrupted computation state without checkpoints. The algorithm is part of a computing model that, to be specific, I called "control flow kernels model in the name of its core component – control kernel. This work describes all the components of the new model, its internal processes, benefits and potential problems.

Keywords: pipeline processing, computations reliability, batch jobs, HPC