СИСТЕМА УПРАВЛЕНИЯ МНОГОШАГОВЫМИ ЗАДАНИЯМИ ДЛЯ СУПЕРКОМПЬЮТЕРОВ И КЛАСТЕРОВ

Ю.А. Чернышов, Н.Н. Попова

Московский государственный университет имени М. В. Ломоносова, факультет Вычислительной математики и кибернетики

Статья посвящена актуальному направлению развития методов и технологий построения прикладного программного обеспечения для суперкомпьютеров. В условиях постоянно увеличивающейся производительности высокопроизводительных систем, роста числа процессорных узлов в таких системах, усложнения постановок решаемых на таких системах проблем становится актуальной задача построения прикладных параллельных программ на основе имеющихся реализаций отдельных подзадач. Данный подход получил название в англоязычной литературе *scientific workflow*.

Понятие *scientific workflow* слабо освещено в русскоязычной научной литературе. Не было введено единого термина для перевода данного словосочетания. *Scientific workflow*, дословно *поток научных задач* — это последовательность задач, выполняющих, как правило, обработку научных данных. Такие потоки задач принято представлять в виде ориентированных графов, вершины в которых соответствуют выполняемым операциям, а рёбра указывают зависимости между операциями, обозначая те ситуации, когда одна задача должна быть полностью выполнена перед тем, как начнётся выполнение другой. Основным требованием к потокам научных задач является их повторяемость при выполнении некоторых заранее определённых требований к архитектуре вычислительной системы и составу доступного программного обеспечения. Результаты выполнения потоков задач должны совпадать на одинаковых входных данных (в том числе, на разных вычислительных системах).

В статье представляются методы построения и приводится описание программной системы (текущее название — *sched*), реализующей поток задач на суперкопьютерах. Описываемая реализация предназначена для работы на системе IBM Blue Gene /P, однако описанные в статье принципы могут применены и к другим вычислительным комплексам под управлением UNIX-like операционной системы.

Описываемая система основывается на следующей модели высокопроизводительной вычислительной системы: вычислительная система — это множество вычислительных узлов (в дальнейшем будем называть их ресурсами), каждый из которых в конкретный момент времени может быть либо занят выполнением программы, либо свободен. Узлами управляет управляющий компьютер с установленным на нём менеджером ресурсов. Взаимодействие с узлами возможно только посредством этого менеджера. Ресурсы объединены в виртуальные очереди, при этом управляющий компьютер, также являющийся ресурсом находится в своей очереди. Планирование задач на управляющем компьютере возможно без участия менеджера ресурсов.

Назовём *командой* пару $\{R, S\}$, где R — требования *команды* к ресурсам вычислительной системы, S — зависимая от системы исполнения строка, задающая способ запуска *команды* на обработку.

Назовём *задачей* тройку $\{C, I, O\}$, где C — непустое множество *команд*, при помощи которых может быть выполнена данная *задача*, I — множество входных наборов данных для этой задачи (возможно, пустое), O — непустое множество наборов данных, создаваемых этой задачей.

Можно дать следующее определения *потока задач*. *Поток задач* — это четвёрка $\{T, D, I, O\}$, где T — множество задач с заранее определёнными требованиями к ресурсам, D — множество наборов данных, $i_k \in I$ — множество входных наборов данных задачи t_k, $o_k \in O$ — множество выходных наборов данных задачи t_k. Верно следующее утверждение: $D = U i_k \cup U o_k$. Каждый набор данных может принадлежать только одному из выходных наборов задачи. Назовём задачу t_j зависящей от задачи t_k, если $\exists d \in D, i \in I, i \in O : d \in i, d \in o$ (задачи называются зависимыми, когда выходной набор данных одной из них пересекается с входным набором данных для другой). Можно построить граф *потока задач*: вершинами графа будут задачи $t_k \in T$, от вершины t_k в вершину t_j ведёт ребро, если задача t_j зависит от задачи t_k. Стоит отметить, что описываемая в статье формальная модель отличается от классического определения, данного в [4].

Специальное подмножество $D_{input} = D \setminus U o_k$ называется входным множеством графа задач. Это те наборы данных, которые не создаются ни одной из задач, входящих в поток, они подаются на вход *потоку задач*.

Выполнение потока задач осуществляется следующим образом. Все задачи помечаются как невыполненные, все наборы данных, кроме множества D_{input} размечаются как недоступные, данные из

множества D_{input} помечаются как доступные. Выполнение состоит из последовательности рабочих циклов. Каждый рабочий цикл состоит из нескольких этапов:

1. Ищется задача t_k, для которой все наборы данных из i_k помечены как доступные.
2. Через обращения к API менеджера ресурсов вычислительной системы запрашивается информация о запущенных и запланированных в данный момент задачах.
3. Уточняются объёмы ресурсов, которые будут выделены задаче для выполнения, выбирается одна из команд задачи, наиболее подходящая для выполнения в данный момент времени.
4. Далее в множестве ресурсов выделяется (ожидается в случае необходимости) необходимое для задачи множество ресурсов.
5. Задача ставится на выполнение.
6. По завершению задачи задача помечается как выполненная.
7. Все наборы данные из o_k помечаются как доступные.

Ожидание завершения выполнения задач (пункт 4) осуществляется асинхронно. Выполнение потока задач завершается, когда все задачи потока помечены как выполненные. На каждой итерации цикла одна из задач помечается как выполненная, и, поскольку число задач в исходном потоке конечно, описанный выше цикл также конечен.

Исполнение потоков задач на кластерах и грид-системах требует поддержки со стороны менеджера ресурсов или стороннего программного обеспечения. На сегодняшний момент не существует ни одного системного планировщика задач (OpenPBS, Torque, LoadLeveler, Slurm), поддерживающего выполнение *scientific workflow*. Обзор программных систем, поддерживающих выполнение потоков задач дан в [1].

Описываемая в статье реализация написана на языке программирования Python (тестировалась с версиями 3.2–3.4, 2.x не поддерживается), некоторые части реализованы на языке C++. В качестве минимальной исполняемой единицы (вершины графа workflow) был выбран процесс. Выбор процесса (в отличие от принятых в некоторых реализациях класса, функтора или функции) позволяет использовать внутри потока задач уже существующие MPI-программы и повышает отказоустойчивость системы: ошибка в одном из компонентов одного из потоков не помешает выполнению других задач.

Система исполнения реагирует на завершение запланированных заданий и следит за тем, чтобы число используемых ресурсов не превышало общий объём ресурсов системы. Для работы системы необходима реализация четырёх функций:

1. получение списка доступных ресурсов,
2. получения состояния очереди (запущенные и запланированные задачи),
3. создание новой задачи с известными требованиями к ресурсам,
4. получение сигнала о завершении задачи.

В текущей реализации в качестве данных могут использоваться файлы и папки, размещённые на общей файловой системе, доступной как на управляющем компьютере, так и на вычислительных узлах кластера. Каждый файл или папка должен обладать уникальным (в рамках одного потока задач) идентификатором.

Граф задач описывается в формате XML в виде двух списков: списка наборов данных и списка исполняемых команд. Команда является шаблоном, в котором могут быть указаны идентификаторы наборов данных, которые будут заменены на абсолютные пути к этим данным непосредственно перед выполнением команды. Использование шаблонизированной командной строки позволяет сделать граф workflow независимым от фактического расположения на файловой системе, становится важным лишь относительное расположение используемых файлов.

Шаблонизированная командная строка выглядит следующим образом (используемая команда — часть пакета программ *Montage* [6], о самом пакете — ниже):

```
mAdd -p $projected_dss2b_folder $projected_dss2b_table $header $joined_dss2b
```

Аргументы, начинающиеся со знака доллара ($) будут заменены на пути к данным.

На данный момент поддерживаются UNIX-like SMP-системы и кластер IBM Blue Gene /P. О перспективах поддержки других платформ будет сказано в конце статьи.

Исходный код системы (а также примеры графов задач в описанном выше формате) можно скачать из GitHub-репозитория по адресу: https://github.com/georgthegreat/sched.

Опишем некоторые особенности реализации и отличительные черты функционала реализованной системы.

Существует два кардинально различных подхода к планированию задач в рамках вычислительной системы: статическое (ресурсы, которые будут выделены для выполнения каждой задачи определяются внешним пользователем на этапе её постановки в очередь) и динамическое (ресурсы определяются непосредственно перед началом выполнения задачи, исходя из информации о свободных ресурсах системы.

Статическое планирование проще в реализации, динамическое позволяет добиться потенциально большей пропускной способности вычислительной системы.

Поскольку большинство существующих систем предназначены для грид-архитектур (например, к этому семейству относится активно развивающаяся система управления потоками Pegasus [2]), существующие

системы обычно реализуются исходя из наличия эксклюзивного доступа к ресурсам (например, такой подход использован в [3]). Однако, поскольку целевой платформой описываемой системы являются кластеры (возможно, с ограниченным набором системных вызовов, как IBM Blue Gene), реализация с использованием эксклюзивного доступа к ресурсам не представляется возможной.

Исходя из необходимости работы в условиях постоянно изменяющейся загрузки вычислительной системы, была выбрана технология динамического планирования.

Большинство MPI-задач могут выполняться не только на каком-то наперёд заданном числе узлов, но и на некотором интервале. При указании команды для запуска на кластере, пользователь указывает значения минимального (mincpus) и максимального (maxcpus) размера партиции, на которую будет назначена задача (можно указать одинаковые значения).

Система автоматически выбирает число узлов, на котором будет запущена задача, исходя из состояния глобальной очереди задач (для всего кластера), а также внутренней очереди, которая используется внутри системы планирования.

Использование динамического планирования позволяет алгоритмизировать некоторые действия, ранее не предусматривавшие изменяющегося поведения. Так, в систему была внедрена базовая поддержка разделяемых задач (divisible task, [5]). Под разделяемой задачей будем понимать параллельные программы, число которых определяется автоматически по размеру входных данных.Папка, в которой находятся входные данные команды, может быть проинтерпретирована не только как один набор данных, но и как набор файлов, содержащихся в этой папке. Как только процесс выполнения дойдёт до задачи, которая помечена как разделяемая, будет принято решение о возможности разделить задачу и запустить по одной задаче на каждый файл из обрабатываемой папки. Существенным ограничением является возможность деления только по одной из входных директорий. В качестве выходного набора данных может быть указана как папка, так и шаблон-файл, имя которого будет вычислено непосредственно перед выполнением.

Пример разделяемой команды (используемая программа — часть пакета программ *Montage*):

mProjectPP #raw_dss2b_folder #projected_dss2b_folder $header

Разделяемые команды обычно запускаются на меньшем объёме вычислительных ресурсов, чем аналогичные команды без разделения. Это позволяет ещё больше повысить пропускную способность системы планирования.

Как уже отмечалось выше, система поддерживает различные способы достижения результата — важно лишь, чтобы запускаемая команда создавала правильные наборы данных при завершении. Логичным шагом было возложить выбор непосредственной команды, которая будет выполнена на систему планирования.

При описании потока задач пользователь может указать одну или несколько команд, которые могут взаимозаменять друг друга. Команды могут быть предназначены как для локального исполнения (т. е. на управляющем компьютере вычислительной системы), так и для удалённого. В процессе работы *sched* постарается автоматически определить наиболее подходящую для исполнения команду, основываясь на информации о свободных вычислительных ресурсах. Так, например, можно одновременно указать две локальные команды, одна из которых будет разделяемой по файлам входной директории, а вторая — нет. Вот примеры эквивалентных командных строк (оба исполняемых файла входят в пакет программ *Montage*):

1. mProjExec -p $raw_dss2ir_folder $raw_dss2ir_table $header $projected_dss2ir_folder $stats_dss2ir_table
2. mProjectPP #raw_dss2ir_folder #projected_dss2ir_folder $header

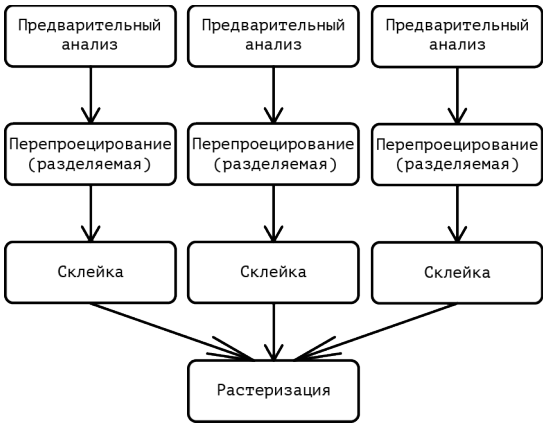

Рис. 1. Граф задач пакета программ Montage.

Тестирование системы проводилось в двух режимах: локального тестирования и тестирования в режиме MPI. Для локального тестирования использовался пакет программ *Montage* (http://montage.ipac.caltech.edu). Набор независимых утилит позволяет выполнять базовые операции по поиску, скачиванию, перепроецированию и растеризации астрономических данных. В ходе экспериментов использовался граф задач, изображенный на *рисунке 1*.

На вход подаются три папки, содержащие предварительно скачанные файлы в необработанном формате (.fits.gz), на выходе — одно изображение в формате JPEG. Хотя время работы графа сильно зависит от скорости ввода-вывода, тесты показали ускорение в 30% по сравнению с запуском при помощи shell-скрипта. Основное ускорение было достигнуто за счёт большего количества задач, запускаемых одновременно. Ресурсы вычислительной системы использовались более эффективно. Стоит отметить, что другие методы ускорения работы программы без изменения исходного кода дают существенно меньший прирост скорости [7].

Тестирование планирования в режиме MPI было проведёно на примере разделяемой задачи обработки данных: поиска минимума и максимума значений в наборе бинарных данных. Сравнивалась время получения результата в двух вариантах запуска: запуск одной задачи с высокими требованиями к ресурсам (1024 вычислительных узла, 4 гигабайта входных данных) и запуск множества (16) задач с меньшими требованиями (32 вычислительных узла, по 256 мегабайт входных данных на запуск).

Запуски проводились в рабочем режиме системы Blue Gene /P, при котором счётным задачам выделяются основные ресурсы, а политики планирования запрещают одновременное выполенние 3 и более задач одного пользователя. Ожидание освобождения большого объёма ресурсов занимало до 10 часов. Результаты проведения эксперимента представлены в таблице 1.

Таблица 1. Результаты тестового запуска в режиме MPI

	Время постановки задачи в очередь (минимум)	Время получения результата (максимум)	Длительность ожидание результата	Длительность выполнения задачи (в среднем, секунд)
1024 узла, 1 запуск	13:12:00	13:32:11	00:22:11	0.674733
32 узла, 16 запусков	13:12:00	13:27:54	00:15:54	0.4717

Из таблицы видно, что разделение задачи уменьшило время получения результата. Хотя данный пример и является искусственным, моделируемая ситуация типична для задач обработки большого объёма данных. Увеличение числа одновременно запускаемых пользователем «небольших» задач позволит сильнее сократить время ожидания результата.

Предлагаемый в статье подход позволяет значительно повысить эффективность использования ресурсов суперкомпьютера за счёт динамической адапции к объёму доступных ресурсов, и, вместе с тем, уменьшить общее время получения результата задачи.

Работа над системой планирования продолжается.

В ближайшее время планируется расширить набор поддерживаемых систем планирования, поддержав менеджер *Slurm*, а также добавить поддержку GPGPU узлов. Планируется также добавить способ задания разделяемых задач для поддержки многовариантных расчётов.

Данная работа использует пакет программ *Montage,* разработанный при поддержке Национального управления по воздухоплаванию и исследованию космического пространства. *Montage* поддерживается NASA/IPAC Infrared Science Archive.

Работа выполнена при поддержке грантов РФФИ № 14-07-00628 и 14-07-00654.

ЛИТЕРАТУРА:

1. Ю. А. Чернышов, Н. Н. Попова Современные подходы к обработки scientific workflow// Программные системы и инструменты. №14. М.: Издатльский отдель факультета ВМиК МГУ. — 2013.
2. Ewa Deelman Grids and Clouds: Making Workflow Applications Work in Heterogeneous Distributed Environments // International Journal of High Performance Computing Applications. — 2010. — url:http://hpc.sagepub.com/content/24/3/284
3. Callaghan S. [et al.] Scaling Up Workflow-Based Applications // Journal of Computer and System Sciences. — 2009. — url: http://pegasus.isi.edu/publications/2010/CallaghanSScalingUp.pdf
4. Chen Q., Wang L., Shang Z. MRGIS: A MapReduce-Enabled High PerformanceWorkflow System for GIS\
5. Xuan Lin, Ying Lu, Jitender Deogun, Steve Goddard Real-Time Divisible Load Scheduling for Cluster Computing. — 2007. — url: http://www.cse.unl.edu/~ylu/papers/rtas07.pdf
6. Software Design Specification for Montage. An Astronomical Image Mosaic Service for the National Virtual Observatory. Version 2.1 (August 29, 2004): Detailed Design. — 2004. — url: http://montage.ipac.caltech.edu/publications/software/Montage_Design_2.13.pdf
7. Vitus J. Leung, David P. Bunde, Johnathan Ebbers, Nicholas W. Price, Matthew Swank, Stefan P. Feer, Zachary D. Rhodes Task Mapping for Non-contiguous Allocations — 2013.